10 de Marzo de 2008

Seminario Hibernate

Motor de persistencia J2EE. Versión 3.2

ARQUITECTURA DE CINCO CAPAS

Nombre de la capa	Responsabilidad	Implementación tecnológica
Presentación	Interfaz de usuario	JSP/HTML/Javascript Swing, SWT, Eclipse RCP Flex (Adobe)
Aplicación	Flujo de navegación, validación sintáctica, interactuación con la capa de servicio	Servlets, Jakarta Struts, Spring MVC, WebWorks, Tapestry, JSF
Servicios	Control de transacciones, flujo de lógica de negocio, rol de fachada	EJB sesión sin estado. POJO + AOP
Dominio	Modelo de dominio, lógica del dominio de negocio, validación semántica	POJO
Persistencia	Persistencia de objetos de dominio	Hibernate, Ibatis, Java Persistence API (JPA), TopLink

Object Relational Mapping. Características

- Existe una diferencia entre el modelo **entidad-relación** propio de los sistemas Gestores de BBDD y el modelo de objetos. Uno es **orientado a sentencia** (SQL) mientras que otro es **Orientado a objetos**.
- Solución: **Enriquecer el modelo de clases Java con información acerca del modelo entidad-relación subyacente**
- Con Hibernate la solución será totalmente **independiente del SGBD**. Independencia del fabricante.
- **Reducción de gran cantidad de código. Mayor fiabilidad**. (con JDBC el código de persistencia se acerca al 30% del código de las aplicaciones).
- Los clientes de la capa de persistencia están totalmente **aislados** del modelo entidad-relación asociado.
- **Permite un mayor rendimiento gracias a su sistema de caché**
- **Permite navegabilidad en el modelo. Relaciones bidireccionales.**
- **Herencia de entidades.**
- Problema de la identidad. Diferencia entre la identidad Java y la identidad en SQL (PRIMARY KEY). ***Surrogate Keys***.

ORM vs EJB's de entidad

- La solución CMP **no permite riqueza de tipos de Java**.
- **No soporta asociaciones polimórficas**, característica importante en *ORM*.
- **Portabilidad discutida**. Las soluciones son bastante dependientes de los fabricantes.
- **No son serializables**. Son necesarios crear objetos ***DTO* o *Value Object* para que la información pueda fluir por las distintas capas de la arquitectura**.
- EJB se considera un **modelo intrusivo**. Hace **complicado la reutilización de código** fuera del contenedor. Provoca **problemas** a la hora de realizar **pruebas unitarias y *Test Driven Development***.

ORM. Estructura

- Una solución ORM contiene los siguientes elementos:
  - **API para realizar operaciones básicas** en objetos persistentes.
  - **Lenguaje de consulta que ataca directamente a las clases y propiedades del modelo** (HQL en caso de Hibernate).
  - Facilidad para **definir metadatos** correspondientes al mapping (enlace BBDD-clases Java).
  - Técnicas que permitan interactuar con **objetos transaccionales**, permitiendo ***dirty checking***, **asociaciones perezosas** (*lazy*) y otras funciones de optimización.
  - **Estrategias de obtención de datos pertenecientes a una asociación. Problema de las N+1 select.**

Ventajas del uso de Hibernate

- **Productividad**: **Evita mucho del código farragoso de la capa de persistencia**, permitiendo centrarse en la lógica de negocio. Permite una **estrategia de desarrollo de aplicaciones** *top-down* (empezar con el modelo de entidades) o *bottom-up* (trabajar con un modelo de datos existente).
- **Mantenibilidad**: Al tener pocas líneas de código permite que el código sea más **comprensible**.
- **Rendimiento**: Existe la tendencia a pensar que una solución "manual" es más eficiente que una "automática". Hay que tener en cuenta que una solución automática, **permite que dediques más tiempo a optimizaciones**. Por otra parte, por ejemplo el pooling de "*PreparedStatement*" son más óptimos para un driver *DB2*, mientras que menos para Interbase. Actualizar las columnas que cambian en una sentencia update es más rápido en unas bases de datos, pero más lentas en otras. Todo está lógica está embebida en el motor *ORM*. **El motor está desarrollado por programadores con altos conocimientos** de los *SGBD* y la conectividad con Java (*JDBC y drivers*).
- **Indepencia de vendedor**: **Una solución ORM te abstrae del SGBD**. Permite desarrollar en local con bases de datos ligeras sin implicación en el entorno de producción.

Soluciones ORM: Hibernate, EJB3 y JPA

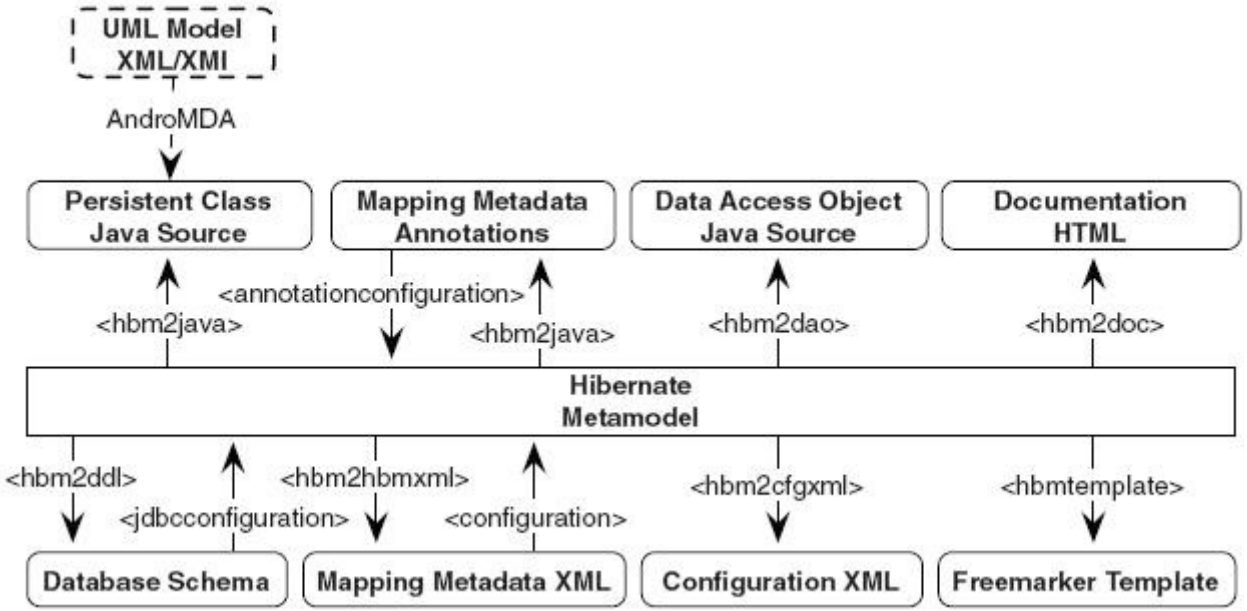
- Hibernate nace a finales del 2001.
- Paralelamente hay una corriente que anima a mejorar la especificación EJB2.1, puesto que la capa de persistencia es mejorable.
- Aparece la JSR 220 (EJB3.0) a principios del 2003. Incluye:
  - Un nuevo modelo de programación de *EJB's* (entidad y *MDB*).
  - Una propuesta nueva para la capa de persistencia: *Java Persistence API*.
- EJB3.0 forma parte del estándar Java EE 5.0.
- Estándares:
  - El motor de JPA debería ser *plugable*, permitiendo cambiar de implementación sin necesidad de cambiar de contenedor.
  - El motor de JPA debe poder correr de manera *standalone* (fuera del servidor de aplicaciones).
  - **Hibernate implementa JPA**. **Se convierte en una implementación de referencia de JPA.**

Módulos de Hibernate

- **Hibernate Core**:
  - Conocido como Hibernate 3.2.x. Corresponde con el servicio base de persistencia con una API propia y **los ficheros de mapping residen en XML.**
- **Hibernate annotations**:
  - Permite definir anotaciones disponibles en JDK5.0 que son **embebidas directamente en el código Java** y evitando disponer de ficheros XML de mapeo.
  - Las anotaciones son un conjunto de anotaciones básicas que implementa el estándar JPA, y además incluye un **conjunto de extensiones que dan cabida a funcionalidades más avanzados propias de Hibernate** (*tunning* y *mapping*).
- **Hibernate EntityManager**
  - La especificación JPA define un conjunto de interfaces, reglas para el ciclo de vida de una entidad persistente y características de las consultas. La implementación de Hibernate para esta parte de la especificación de JPA es cubierta con Hibernate EntityManager. **Las características de Hibernate son un superconjunto de las especificadas por JPA.**
- Para determinar si la solución es **compatible con JPA** todas las importaciones deben ser del **paquete javax.persistence**. En el caso que haya alguna importación del **paquete org.hibernate** la solución **no será compatible JPA.**
- El grupo de desarrolladores de **Hibernate** se mantiene aparte de la especificación con la idea de **dar rápida respuesta a las necesidades futuras**. Estas nuevas funcionalidades serán **candidatas para la elaboración de una nueva versión del estándar.**

Proceso de desarrollo

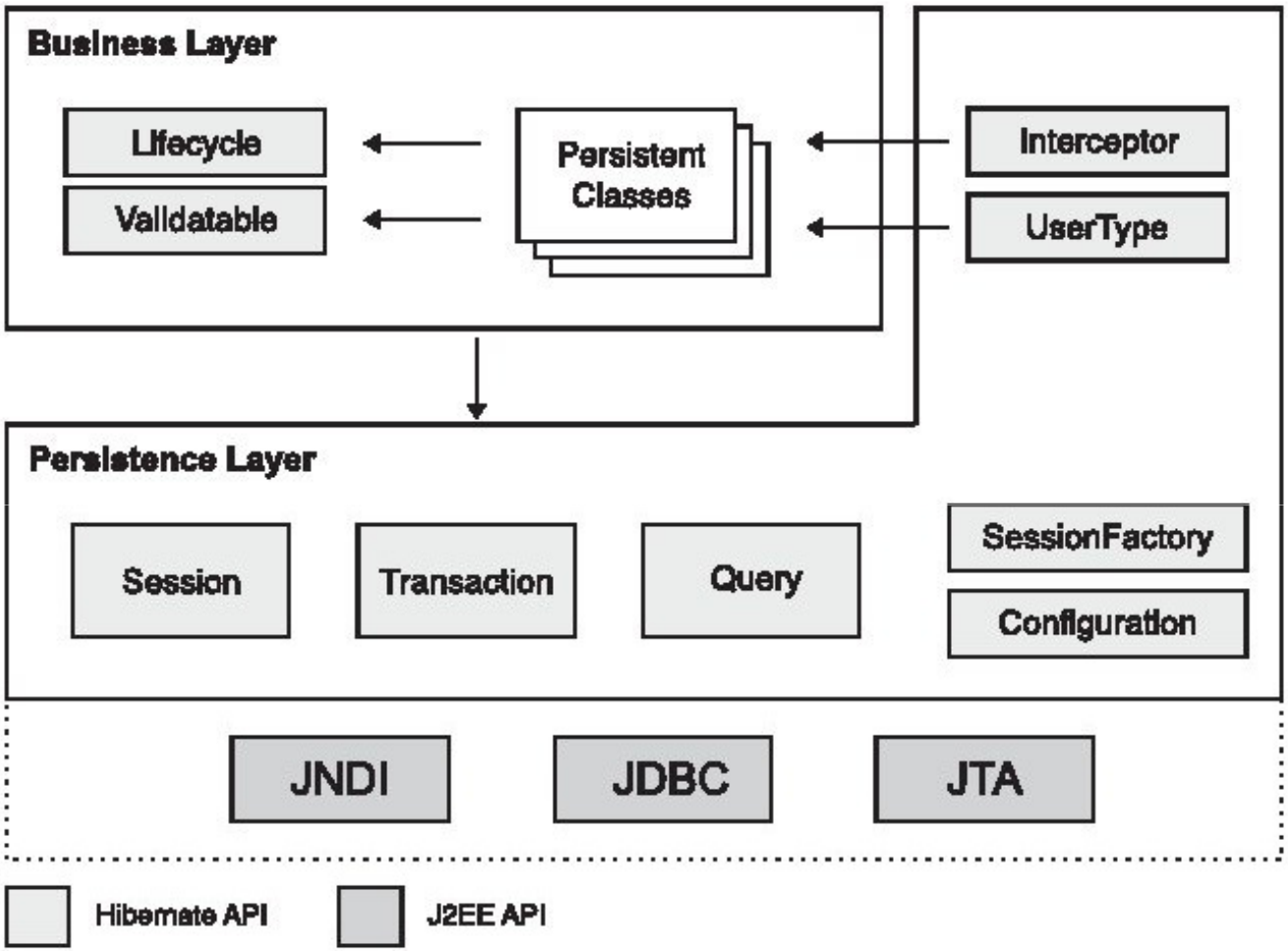
- ***Bottom-up***. **Se parte de un modelo de datos** y mediante la utilidad **hbm2hbmxml se generan** los ficheros de mapping. Hay cierta información que no es posible extraer con esta herramienta. Posteriormente, es posible ejecutar **hbm2java** para generar las entidades de dominio.
- ***Top-down***: Comenzamos con el desarrollo del modelo de entidades y el fichero de mapeo, y generamos el script de la BBDD a partir de estos dos componentes (**hbm2dll**)
- ***Middle-out***: Se **desarrollan los ficheros de mapeo directamente** y se generan las clases del modelo de entidades y el script de la BBDD.
- En el caso que tanto la BBDD como las clases del modelo de entidades sea heredada, **no existe herramienta que te ayude**.



Paradigma

ARQUITECTURA BÁSICA

Arquitectura básica



Paradigma

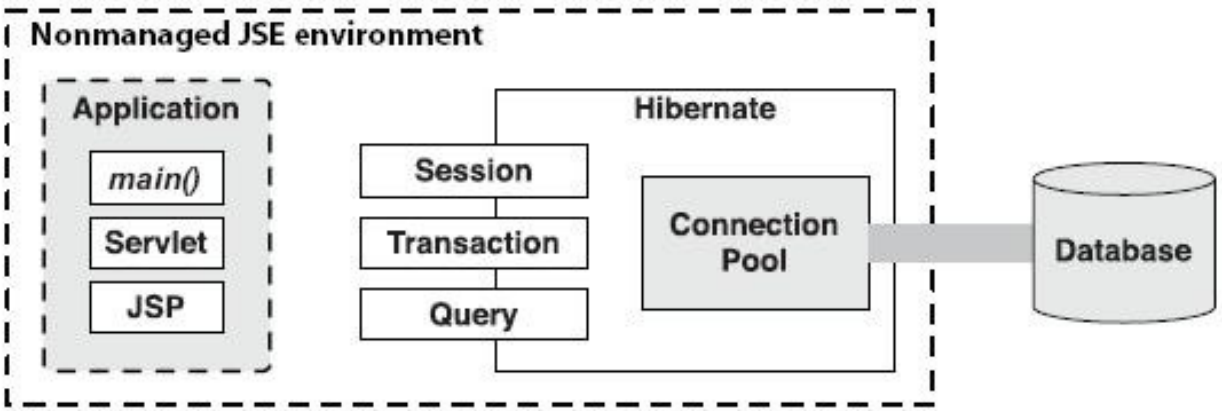
Componentes básicos

- **Session**: Corresponde con un objeto que representa **una unidad de trabajo con la base de datos (transacción)**. Además representa el **gestor de persitencia**, ya que dispone de la API básica para poder cargar y guardar objetos. Internamente **consiste de una cola de sentencias SQL** que son necesarias ejecutar para sincronizar el estado de la sesión con la BBDD. Además contiene una lista de objetos persistentes. **Corresponde con un primer nivel de caché**.
- **Transaction**: La API de Hibernate contiene utilidades para **demarcar la transaccionalidad** de operaciones de **manera programática**.
- **Query**: Este interfaz permite **crear consultas y enlazar argumentos a parámetros de la consulta** (binding). Permite definir consultas en HQL (Hibernate Query Language) o en SQL.
- **SessionFactory**: Es una **factoría de sesiones**. Proporciona objetos Session. **Es thread-safe**. **Permite concurrencia**.
- **Configuration**: Encargado de **cargar los ficheros de mapping**, las **propiedades específicas de Hibernate** y entonces crear el *SessionFactory*.

Consideraciones al ejemplo HelloWorld

- **Automatic Dirty checking:**
  - Los objetos persistentes que son actualizados dentro de un contexto transaccional (unidad de trabajo), hacen que automáticamente se lance las sentencias de actualización de la BBDD **sin llamar explícitamente al método update**.
- **Persistencia transitiva:**
  - Permite que los **objetos de una asociación sean insertados, actualizados o borrados** sin necesidad de realizarlo explícitamente
- **Escritura retardada**
  - Hibernate usa un algoritmo sofisticado que determina un **orden eficiente al ejecutar las sentencias (todos los objetos del contexto de persistencia que han cambiado),** evitando la violación de restricciones de clave foránea.

Entorno no gestionado

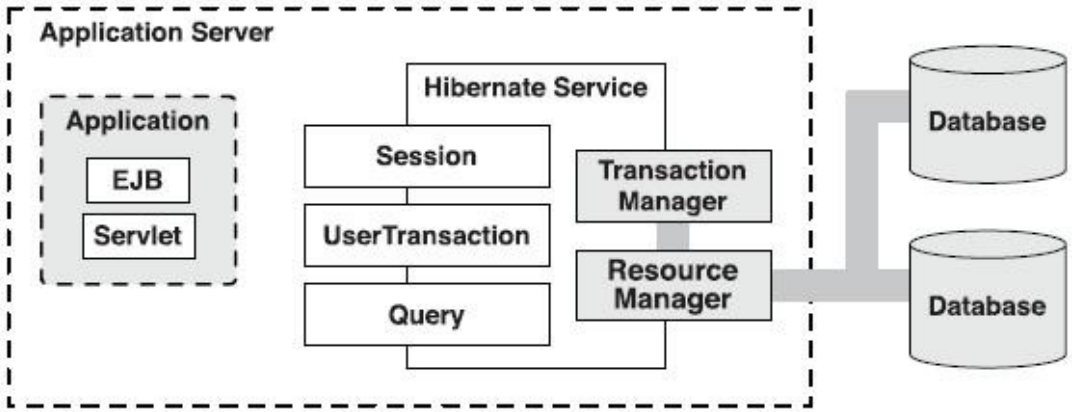



```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
```


Paradigma

Entorno gestionado

- La gestión del pool de conexiones, de la transacción y de los recursos es delegada al servidor de aplicaciones




```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```


Paradigma

Configuración. Guía rápida

- Por defecto el componente **SessionFactory** va buscando un recurso llamado ***hibernate.cfg.xml*** dentro del *ClassLoader* (cargador de clases) .
- Dentro del fichero ***hibernate.cfg.xml*** se configura:
  - **Listado de ficheros de mapeo**
  - **Dialecto de la BBDD**. Engloba aquellas particularidades del SGBD.
  - **Cadena de conexión a la BBDD**.
  - Propiedades del **pool de conexiones** (C3PO, Apache DBCP, …)
  - Propiedades adicionales:
    - ***show_sql***: Muestras las sentencias SQL emitidas por el motor de persistencia
    - ***format_sql***: Formatea las cadenas de las consultas para que sean legibles.
- Una vez obtenido el *SessionFactory* (proceso costoso), por cada petición al motor de persistencia, se obtiene un **componente Session** que se convierte en el interfaz para cargar y persistir objetos.

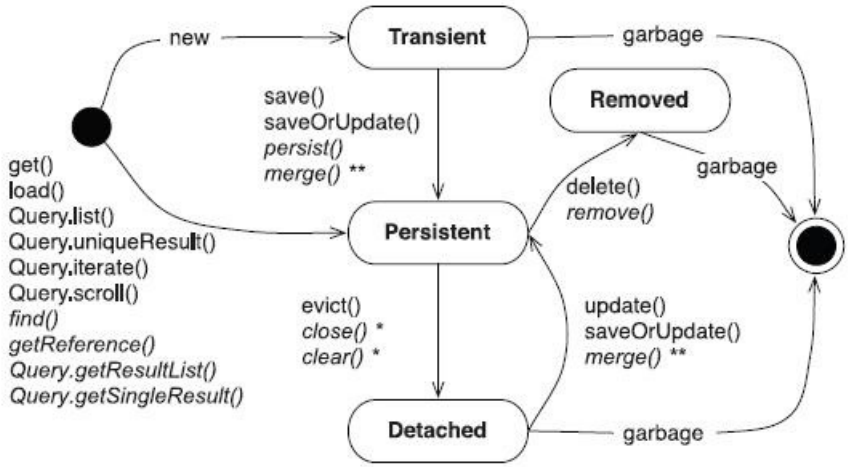
Anotaciones

- Las anotaciones permiten definir los **metadatos en el mismo fichero fuente Java** evitando que la información resida en un fichero XML (fichero de mapeo).
- Las anotaciones **son procesadas** en el **tiempo de arranque del SessionFactory** y toda la información de mapeo está disponible para el motor de persistencia.
- **Evita incompatibilidades** entre el fichero de mapping y el objeto de dominio ya que toda la información reside en el mismo fichero físico.
- Dentro del fichero de configuración de hibernate es **necesario informar que el fichero de mapeo es una clase y no un recurso**.

TRABAJANDO CON OBJETOS

Ciclo de vida de objetos persistentes

- Puesto que Hibernate es un mecanismo que **permite dar solución a la capa de persistencia de un modo transparente**, los **objetos que son persistidos no se deben preocupar de los estados por los que pasa. Es Hibernate el que maneja estos estados**.
- Hibernate define 4 estados, ocultando de la complejidad de su implementación al usuario del motor.



Paradigma

Comenzando una unidad de trabajo

Session session = sessionFactory.openSession();

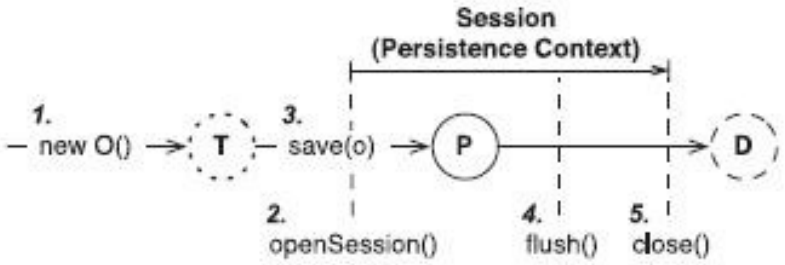
Transaction tx = session.openTransaction();

- En este punto el **contexto de persistencia ha sido inicializado**.
- La aplicación **puede tener varios SessionFactory**, cada uno conectado a una Base de Datos
- La **creación de SessionFactory es costosa**, con lo que se aconseja inicializar el SessionFactory en el **arranque de la aplicación una única vez**.
- La **obtención de Session es muy ligera**, de hecho **una Session no obtiene la conexión JDBC hasta que no es necesario**.
- En la **última línea se abre una transacción**, y todas las operaciones que se ejecuten dentro de la unidad de trabajo se realizarán en la misma transacción.

Persistiendo un objeto

- 1. **Se instancia un objeto nuevo (estado transient).**
- 2. **Se obtiene una sesión y se comienza la transacción**, inicializando el contexto de persistencia
- 3. Una vez obtenida da la sesión, se llama al método save(), **el cual introduce el objeto en el contexto de persistencia**. Este método **devuelve el identificador del objeto persistido**.
- 4. Para que los **cambios sean sincronizados** en la BBDD, es necesario **realizar el commit** de la transacción. Dentro del objeto **sesión** se llama al **método flush().** Es posible llamarlo explícitamente. En este momento, **se obtiene la conexión JDBC a la BBDD** para poder ejecutar la oportuna sentencia INSERT.
- 5. Finalmente, la **sesión se cierra**, con el objeto de **liberar el contexto de persistencia**, y por tanto, **devolver la referencia del objeto creado al estado detached**.

```
Item item = new Item();                                          ❶
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );

Session session = sessionFactory.openSession();                  ❷
Transaction tx = session.beginTransaction();

Serializable itemId = session.save(item);                        ❸

tx.commit();                                                     ❹
session.close();                                                 ❺
```

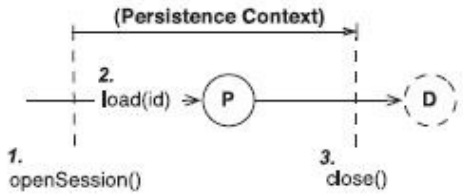


Paradigma

Recuperando un objeto persistente

- Existen dos **métodos que se encargan de recuperar un objeto persistente por identificador**: *load()* y *get().*
- La diferencia entre ellos radica en cómo indican que un objeto no se encuentra en la base de datos: ***get()*** **devuelve un nulo y *load()* lanza una excepción ObjectNotFoundException**.
- Aparte de esta diferencia, load() intenta devolver un objeto proxy siempre y cuando le sea posible (no esté en el contexto de persistencia). Con lo que es posible que la excepción sea lanzada cuando se inicialice el objeto proxy. Esto es conocido como **carga perezosa**.
- En el caso que se **obtenga un proxy, no tiene impacto sobre la base de datos** (no se ejecuta ninguna consulta), hasta que no se inicializa el mismo.
- Muy útil **cuando se obtiene una referencia de un objeto para asociarlo a otro**. (No es necesario obtener el objeto).Se modifica un objeto persistente.

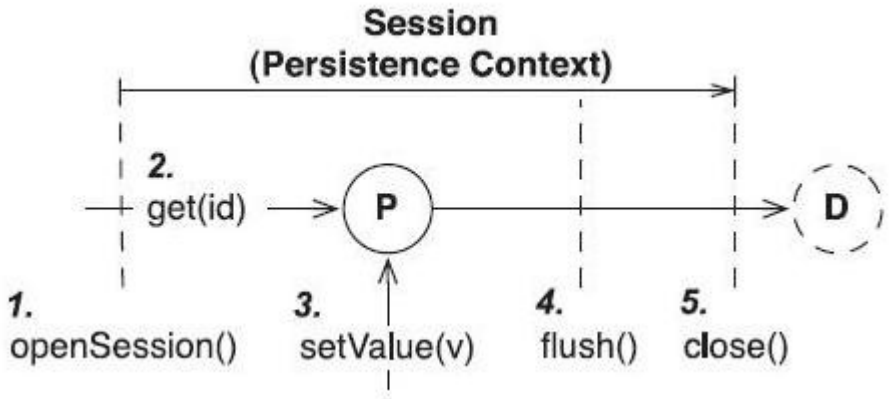
```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.load(Item.class, new Long(1234));
// Item item = (Item) session.get(Item.class, new Long(1234));

tx.commit();
session.close();
```



Paradigma

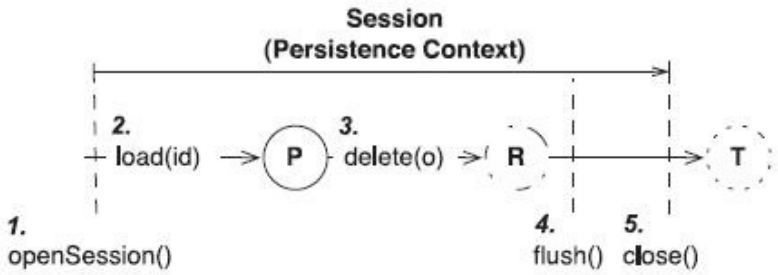
```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.get(Item.class, new Long(1234));

item.setDescription("This Playstation is as good as new!");

tx.commit();
session.close();
```

NO SE LLAMA A UPDATE EXPLÍCITAMENTE!!!!!



- Para poder borrar un objeto **es necesario obtenerlo previamente** (mediante proxy mejor). Esto es debido a que en Hibernate es posible **tener un conjunto de Interceptores**, y el objeto debería de **pasar por esos interceptores para completar su ciclo de vida**.
- Hibernate ofrece la posibilidad de ejecutar operaciones masivas sin necesidad de recuperar la instancia del objeto (**Bulk and batch operations**)

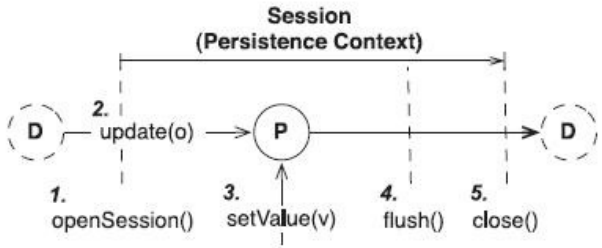
```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

Item item = (Item) session.load(Item.class, new Long(1234));

session.delete(item);

tx.commit();
session.close();
```



Paradigma

Convirtiendo un objeto detached en persistente (Reattaching)

- En este caso Hibernate **no detecta automáticamente que se pretende actualizar un objeto detached**. **Es necesario llamar explícitamente al método update del objeto Session**.
- Hibernate trata el objeto como sucio y planifica la ejecución de una sentencia UPDATE.
- Se puede **evitar la ejecución de la sentencia UPDATE** si en el fichero de mapeo se específica el valor ***select-before-update="true"***. En este caso, Hibernate determina si el objeto está sucio lanzando una sentencia SELECT.




```
item.setDescription(...); // Loaded in previous Session

Session sessionTwo = sessionFactory.openSession();
Transaction tx = sessionTwo.beginTransaction();

sessionTwo.update(item);

item.setEndDate(...);

tx.commit();
sessionTwo.close();
```


Paradigma

Detached →Transient

- En el ejemplo, se planifica una vuelta al contexto de persistencia, para luego, mediante la llamada al método delete() planificar el borrado del objeto pasando al estado transient.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

session.delete(item);

tx.commit();
session.close();
```

Caché del contexto de persistencia

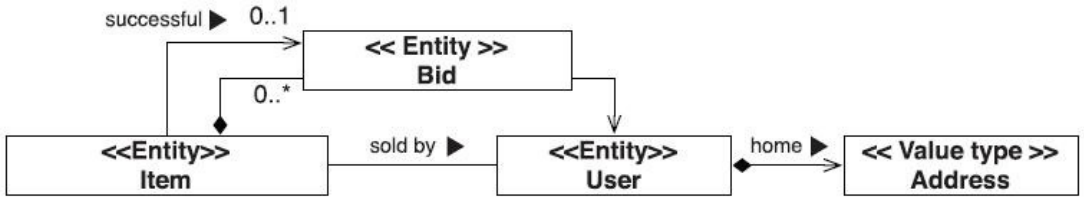
- El contexto de persistencia es una **caché de objetos persistentes**. Por tanto, el contexto de persistencia sabe **de todas las instancias de los objetos en estado persistente**, y además maneja una referencia a un "*snapshot*" de cada uno de ellos, con el objeto de implementar internamente **la funcionalidad de dirty-checking.**
- Esto provoca que para casos de **operaciones masivas de actualización**, se realice un **uso excesivo de esta caché** y pueda conducir en una **falta de memoria de la máquina virtual**.
- Asimismo, en el caso que los grafos de **objetos no estén bien configurados** (asociaciones no perezosas), se **llena la caché con elementos innecesarios.**
- **La caché no disminuye automáticamente** dentro de una unidad de trabajo.
- Para poder reducir el tamaño de la caché de objetos se puede:
  - Llamar al método ***evict(object),*** el cual **pasa el objeto a estado detached**, con el objeto que no se mantenga una copia en la cache.
  - Llamar al método **session.clear()**, el cual pasa **todos los objetos persistentes a detached**, liberando todas las copias de referencias en la caché.
  - **session.setReadOnly(object, true)**, el cual **deshabilita el control de dirty-checking** para un determinado objeto. Con el segundo parámetro a *true*, se indica que se puede volver a habilitar el control de dirty-checking.
- En el final de la unidad de trabajo, todas las **modificaciones hechas sobre los objetos persistentes**, son **sincronizadas con la base de datos a través de sentencias DML**. Este proceso es conocido como ***flushing.***

Escritura retardada efectiva (write-behind efective)

- Hibernate, retarda la ejecución de los cambios a la BBDD hasta el final de la unidad de trabajo. Esto beneficia en:
  - **Reducción de peticiones a la base de datos**.
  - **Minimiza el impacto provocado por la latencia de red**.
  - **La duración de los bloqueos dentro de la BBDD se reducen.**
- Si una propiedad de un **objeto cambia dos veces**, únicamente **se lanza una sentencia UPDATE**.
- Hibernate se aprovecha de las ventajas de la API de **JDBC batch**, cuando se ejecutan numerosas sentencias DML.
- La **sincronización de la BBDD** con el contexto de persistencia ocurre en los siguientes casos:
  - Cuando la **transacción en la API de Hibernate finaliza** (No en la JDBC).
  - **Antes de ejecutar una consulta**.
  - Cuando la aplicación **llama a *session.flush()*** de manera explícita.
- Existen tres modos de Flushing configurables:
  - **AUTO**: Lo mencionado en el punto anterior
  - **COMMIT**: No se sincroniza cuando se ejecuta una consulta
  - **MANUAL**: Sólo sincronización manual

Entidades y tipos de valor (value type).

- Hibernate diferencia entre dos tipos de entidad. Una ciudadano de primera clase que son las ***entidades*** y otras llamadas ***value type***.
- Ejemplo: La entidad usuario tiene una dirección asociada. Puede ser que dos usuarios residan en la misma dirección. Dos posibles diseños: Los campos asociados a la dirección son asociados a la entidad Usuario con lo que cada usuario tiene su dirección y si es la misma se debe repetir. O por el contrario la dirección tiene la suficiente importancia como para declararla como entidad. En esta última alternativa habrá una única dirección asociada a dos usuarios.
- En el primero de los casos se trata de un *value type* y la identidad viene siempre asociada a la entidad a la que está asociada.
- En el segundo de los casos se trata de una entidad y por tanto tiene su propia identidad.
- Con el objeto de **favorecer la reutilización se anima a disponer de tantas clases como sea necesario** (más que tablas).
- En el ejemplo expuesto: La relación entre Item y Bid con cardinalidad 0..n es una composición (Bid es un value-type) pero la asociación (0..1) hace que Bid deba tener su propia identidad (Entidad) ya que tiene referencias compartidas.



Paradigma

Generadores de identificadores

Generador	Descripción
native	Usa identity, sequence o hilo en función del SGDB residente
identity	Soportado para DB2, MySQL, SQLServer, Sybase, HSQLDB, Informix.
sequence	Secuencia DB2, PostgreSQL, Oracle, SAPDB, McKoi, Firebird
increment	Obtiene el identificador más alto de la tabla y lo incrementa en uno. Eficiente e caso que el servidor de Hibernate (único) tenga acceso único al esquema
hilo	Algoritmo hilo. Tipo numérico. Válido únicamente para una BBDD
uuid.hex	128-bit UUID. Usa la IP y el timestamp para generar la clave. Clave hexadecimal de tamaño 32. Válido para varias instancias del SGBD.

Consideraciones

- **Transitorio:**
  - Un objeto en estado ***transitorio*** **se considera no transaccional**. Por lo tanto, **cualquier cambio** sobre él, **no va a ser monitorizada por el contexto persistente** (Session).
  - **Los objetos que son referenciados por un objeto transitorio, por defecto, son transitorios**. Para pasar de **estado *transitorio* a *persistente* es necesario o llamar al motor de persistencia o que un objeto persistente tenga una referencia al objeto transitorio**
- **Persistente:**
  - Un objeto con estado *persistente* es una **instancia con una identidad de BBDD**. Aparte de lo comentado en el punto anterior **es posible que un objeto pase a estado persistente** porque se **obtenga de la ejecución de una consulta** o **porque se navegue por el grafo de asociaciones de un objeto persistente**.
  - Los **objetos persistentes residen en el contexto de persistencia**, con lo que se **cachean** e Hibernate **monitoriza cualquier cambio** que haya sobre ellos.
- **Objetos borrados**
  - Un objeto en estado borrado es **planificado a ser borrado una vez finalice la unidad de trabajo** (transacción). Por tanto, se **debería descartar cualquier referencia a este objeto**.

Consideraciones (II)

- **Objeto detached**
  - Mientras una ***unidad de trabajo*** **está en curso**, **todos aquellos objetos que están en estado persistente** se encuentran en el **contexto de persistencia**. Sin embargo, cuando **esta unidad de trabajo finaliza**, el contexto de persistencia se cierra, **y todos los objetos de este contexto pasan a detached.**
  - Todos los objetos que están en estado detached **no se garantiza que se vaya a sincronizar su estado con la BBDD.**
  - Sin embargo, en algún punto de la aplicación **puede interesar modificar el objeto y volver a sincronizar su estado**. En otras palabras, pasar del estado detached al estado persistente. Ejemplo: Aquellos objetos que se han renderizado en la capa de presentación.
  - Hibernate ofrece dos operaciones ***reattachment*** **y** ***merging*** para tratar esta situación (JPA sólo merging). Esto permite que, por tanto, **haya unidades de trabajo "duraderas" que se propaguen incluso a la capa de presentación**. Estas unidades de trabajo se conocen como **conversaciones**.

Persistencia transitiva

- Por defecto Hibernate, **no navega por las asociaciones**, con lo que **operaciones de inserción, borrado, modificación, merging y reattaching no tienen efecto sobre las entidades asociadas**.
- Para cambiar el funcionamiento por defecto, Hibernate **permite la configuración de cada una de las asociaciones**. **(Atributo *cascade* en asociaciones many-to-one, *one-to-many* y *many-to-many).***
- A continuación se presentan las opciones que el atribute cascade puede aceptar

Tipos de *cascade*

Atributo XML	Descripción
None	Ignora la asociación
save-update	Hibernate navega la asociación cuando la sesión es sincronizada y cuando el objeto pasa a ejecutar el correspondiente método save() o update(), grabando las nuevas instancias transient de la asocaicón y persistiendo los cambios sobre las instancias detached.
persist	Hibernate persiste cualquier instancia transient asociada cuando el se llama al método persist() del objeto.
merge	Navega la asociación y realiza la operación merge sobre las asociadas instancias detached. La operación se realiza sobre las instancias en estado persistente residentes en el contexto de persistencia. Las instancias asociadas que son transient son cambiadas a estado persistente.
delete	Borra las entidades asociadas en estado persistente
Lock	Incorpora al contexto de persistencia aquellas instancias de entidades asociadas que estén en estado detached. El modo de bloqueo (LockMode) no es propagado.
evict()	Borra de la caché todas las instancias de entidades asociadas
refresh	Se recupera el estado de los objetos asociados de la base de datos
all	Todas las opciones mostradas anteriormente
delete-orphan	Borra las entidades asociadas cuando son eliminadas de la asociación. Se usa cuando la entidad borrada no tiene referencias compartidas

Procesamiento masivo

- Puesto que el contexto de persistencia está habilitado por defecto, provoca que ante **operaciones masivas se necesite una gran cantidad de memoria**.
- Por ejemplo si quisiéramos hacer un proceso masivo de todos los elementos de una tabla, necesitaríamos una cantidad ingente de memoria.
- Soluciones:
  - **Procesamiento en lotes**. Recoger la información en lotes de X elementos (con un cursor en la consulta → ScrollableResult). Consideración: **Establecer el valor de la propiedad hibernate.jdbc.batch_size** al valor del tamaño del lote.
  - Uso de **StatelessSession** No presenta la necesidad de realizar el trabajo en lotes, puesto que no tiene contexto de persistencia asociado.
    - Problemas derivados:
      - **No dirty checking, no write-behind.**
      - **No hay persistencia transitiva**
      - **Se pierde la identidad de objetos en sesión. Cada consulta produce diferentes instancias en memoria de la misma fila.**
      - **No hay Interceptores.**

Optimización en la obtención de objetos

- Hibernate presenta las siguientes formas de obtener objetos:
  - **Navegando por el grafo de objetos** que previamente han sido cargados en el contexto de persistencia.
  - Obtener una **entidad por identificador.**
  - Lanzar **una consulta con HQL** (Hibernate Query Language) que es un lenguaje de consultas orientado a objetos.
  - Usando el **interfaz Criteria para ejecutar sentencias programáticamente**: garantiza el chequeo de tipos y una orientación a objetos de la sentencia.
  - **Ejecutar una sentencia SQL nativa usando la API de JDBC.**
- **Plan de obtención** (**fetch plan**): Define si y como traerse un objeto asociado o una colección.

Hibernate Query Language

- Es usado normalmente **para obtención de información, más que para sentencias DML** (normalmente que se encargue el motor de persistencia).
- Soporta las siguientes funcionalidades:
  - **Restricciones sobre objetos asociados (colecciones)**
  - **Obtener un subconjunto de propiedades de la entidad**. A este concepto se conoce como *projection* o *report query.*
  - **Ordenación de resultados**
  - **Paginación de resultados**
  - **Agregación** mediante *group by*, *having*, y funciones agregadas: *avg, sum, min, max,…*
  - **Outer joins** cuando recuperas múltiple objetos por fila.
  - La habilidad **de llamar a funciones SQL estándar y definidas por el usuario**.
  - **Subconsultas**

Consultas con Criteria

- **Evita la manipulación de cadenas** para la construcción de la consulta.
- **No se pierde el poder del HQL**.
- Muchos desarrolladores la prefieren por ser **una solución con más orientación a objetos**.
- Un *Criteria* está formado por **un árbol de instancias *Criterion***. La clases ***Restrictions*** **provee una factoría estática de métodos que devuelven instancias de instancias *Criterion*.**
- Consulta por ejemplo (*Query by example - QBE*)
  - Construye la sentencia a partir de un objeto que ha sido "seteado" parcialmente. **Genera la cláusula *where* realizando un "match" únicamente con los atributos seteados**.
  - **Muy útil para formularios de búsqueda.**
  - Esta clase de funcionalidad **presenta complejidad en el caso que se quiera expresar con un lenguaje de consultas**.

Asociaciones perezosas. Concepto de proxy

- Por defecto, todas **las entidades y colecciones se cargan mediante una estrategia perezosa**. Es decir, sólo se cargan aquellos objetos que se están pidiendo en la consulta.
- Proxy:
  - Hibernate **crea un *proxy* siempre que puede para evitarse el tener que lanzar un *hit* en la base de datos**.
  - Un ***proxy* consiste en un objeto "apoderado", capaz de actuar por el otro objeto**. Esto provoca que no ejecute el hit en la BBDD hasta que no es accedido el *proxy* por primera vez.
  - **Es muy útil cuando se produce una asociación entre dos objetos** (No es necesario traerse los dos), cuando se carga un objeto que tiene colecciones (no me traigo todos los datos de la asociación, …).
  - **El objeto proxy lo único que contiene es el identificador.**
- Es posible **eliminar esta opción por defecto** estableciendo el atributo **lazy con un valor a false**. Con esto, evitaremos la generación del proxy y se cargará por defecto las entidades o propiedades afectadas.
- Además, existe otra **tercera opción que es usar *Interception* en vez de proxies. (*lazy=no-proxy*).** Es menos perezoso que el proxy.
- Además es posible realizar **la carga perezosa de propiedades**, pero esta solución requiere ***Instrumentación*** *(*modificación del bytecode de la clase con el objeto de meter la lógica de carga de propiedades perezosa en la clase. En Java no hay soporte nativo para tal funcionalidad).

Estrategias de obtención de información

- Preobtención de información:
  - Es **posible inicializar un número prefijado de proxies**, mediante el atributo *batch-size.*
  - La primera vez que se hace referencia a la entidad asociada, Hibernate **inicializa un conjunto de proxies, con el objeto de no lanzar una consulta por cada una de las entidades asociadas**. Problema de las N+1 select → Reduce el número de consultas.
  - **Es posible definirlo a nivel de entidad o a nivel de colección.**
- Obtención de colecciones con subconsultas:
  - **Se indica mediante el atributo *fetch="subselect"* a nivel de colección .**
  - **Con esta estrategia, la primera vez que se accede a los datos de la colección, se inicializan todos los elementos de la colección (por cada una de las entidades raíz obtenidas en la consulta).**
  - **No se permite esta estrategia a nivel de entidad.**
- Obtención de los datos mediante join
  - **Se indica mediante el atributo *fetch="join"***
  - Es válido para asociaciones **one-to-many** y **many-to-one**.
  - Para traerse los datos asociados **ejecuta un join con el objeto de evitar la select adicional**.
  - En las asociaciones **one-to-many siempre se ejecutan outer-join para evitar la no obtención de entidades sin asociación con la entidad hija**.
  - Es posible **limitar el número de joins** con la opción de hibernate: ***hibernate.max_fetch_depth***

Líneas para la optimización de carga de datos

- Por defecto, **Hibernate sólo carga los datos necesarios con lo que se reduce el consumo de memoria del contexto de persistencia**. Sin embargo, se presenta el **problema de las n+1 consultas para la obtención de elementos de una asociación**. Para **evitar este problema** se aconseja **el uso de la estrategia batch o subconsulta**. En el caso que **sea necesario obtener todos los datos de la colección asociada considerar el uso de la estrategia *join*** pero a nivel de consulta/Criteria.
- En el supuesto que se configure la **opción de obtención de datos mediante join**, se puede incurrir en el problema del producto cartesiano, lo **cual provoca que en vez de ejecutar demasiadas consultas, se obtengan un exceso de datos**. Valorar el caso de una entidad que tenga **dos colecciones con estrategia join**. En el caso de colecciones paralelas **se aconseja el uso de subconsultas**.
- Por tanto, es necesario **encontrar un punto medio entre ambos extremos**.
- Si se necesita alterar el contenido de un objeto en estado detached, **es posible en Hibernate inicializar explícitamente la asociación mediante el método Hibernate.initialize(),** con lo que se evita que se lance la excepción *org.hibernate.LazyInitializationException.*