Resumen de conceptos:

Persistencia de datos: Paso de datos de la memoria principal al medio de almacenamiento secundario (disco) para que puedan volver a ser recuperados a la memoria principal.

Objeto persistente: Objeto que tiene una representación persistente en la base de datos. El objeto persistente está asociado a la detección de los cambios que se realizan sobre él para poder reflejarlos en la base de datos.

Desfase objeto-relacional: Los problemas para la persistencia de objetos en ficheros o en bases de datos relacionales.

ORM (Object Relational Mapping): Correspondencia objeto relacional. Son las técnicas y herramientas que hacen posible la persistencia de objetos en un esquema relacional, teniendo en cuenta la correspondencia con los objetos que representan a las clases en el modelo de programación OO.

JPA (java Persistence Arquitecture): Es la API de java para ORM.

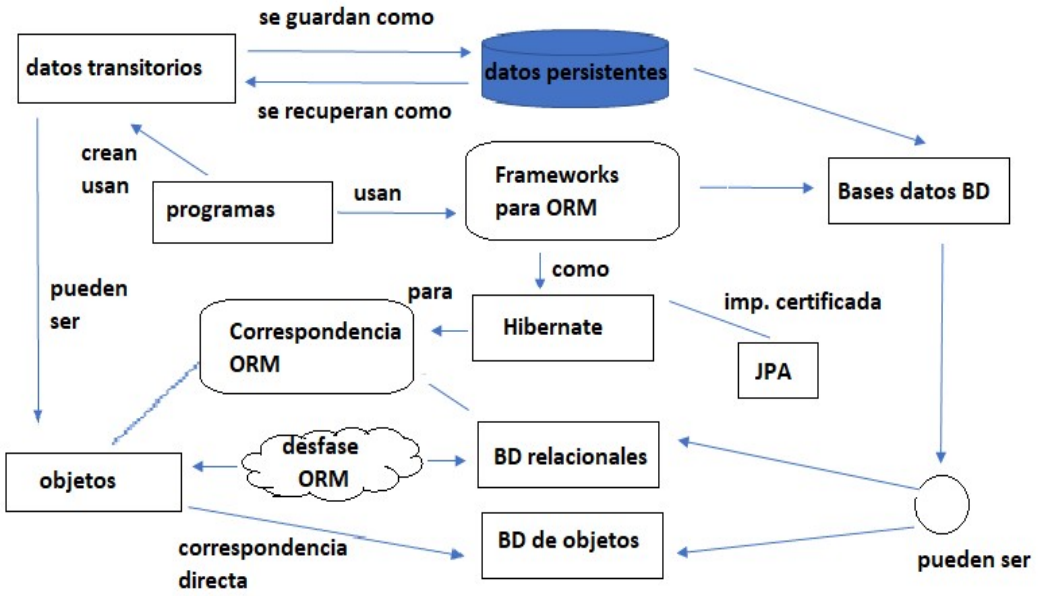
1. ¿Por qué necesitamos Hibernate?

Trabajaremos con programación orientada a objetos y utilizaremos bases de datos (BD) relacionales.

Son dos paradigmas diferentes.

- El modelo relacional trata con relaciones, tuplas y conjuntos, y es muy matemático por naturaleza.

- El paradigma orientado a objetos trata con objetos, sus atributos y relaciones entre objetos.



1. ¿Por qué necesitamos Hibernate?

Cuando se quiere hacer que los objetos sean persistentes utilizando para ello una BD relacional, uno se da cuenta de que hay una desavenencia entre estos dos paradigmas: es lo que se denomina un *"object-relational gap"*.

¿Cómo se manifiesta esta brecha entre ambos paradigmas?

Si utilizamos objetos en nuestra aplicación y queremos que sean persistentes, normalmente abriremos una conexión JDBC, crearemos una sentencia SQL y copiaremos todos los valores de las propiedades sobre una PreparedStatement o en la cadena SQL que estemos construyendo.

Si lo objetos de tipo valor(VO) son pequeños no solemos tener problemas, pero si tienen muchas propiedades…

Otras preguntas que se nos plantean ahora…

¿Qué ocurre con las asociaciones? ¿Y si el objeto contiene a su vez aotros objetos? ¿Los almacenaremos también en la BD? ¿Automáticamente? ¿Manualmente? ¿Qué haremos con las claves ajenas?

1. ¿Por qué necesitamos Hibernate?

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 (Object Relational Mapping).

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 VOs directamente.

1. ¿Por qué necesitamos Hibernate?

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.

Hibernate es una ORM de libre distribución, que además, es de las más maduras y completas.

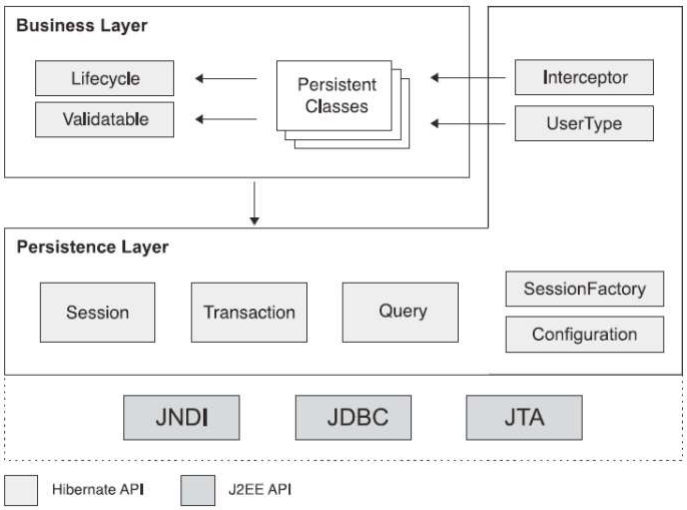
Actualmente su uso está muy extendido y además está siendo desarrollada de forma muy activa.

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.

2. Arquitectura Hibernate

La siguiente Figura muestra los roles de las interfaces Hibernate más importantes en las capas de persistencia y de negocio de una aplicación J2EE.

La capa de negocio está situada sobre la capa de persistencia, ya que la capa de negocio actúa como un cliente de la capa de persistencia.



Correspondencia ORM.

El uso de lenguajes OO para trabajar con bases de datos relacionales plantea problemas conocidos como desfase objeto-relacional. La forma de resolver estos problemas se basa en alguno de estos planteamientos:

1. **Bases de datos Objeto Relacionales**:
   Son bases de datos relacionales con capacidad para gestionar objetos. En SQL99 se incluyeron tipos estructurados definidos por el usuario, que pueden ser tipos de objetos (clases). Sin embargo, resuelven parcialmente el problema.
1. **ORM o correspondencia objeto relacional**.
   Se trata de establecer la correspondencia entre las clases definidas en Java y las tablas definidas en la BD relacional. Así como el mecanismo que permite registrar en la base de datos los cambios introducidos en los objetos y recuperar como objetos la información de la base de datos.

   Esta correspondencia hace posible la persistencia de objetos en bases de datos relacionales.
1. **Bases de datos OO**.
   Almacenan objetos directamente. Aunque parece la mejor solución, lo cierto es que este tipo de bases de datos apenas se usa debido a la potencia y fuerza con la que se empezaron a implantar las bases de datos relacionales.

Hibernate.

URL: https://hibernate.org/

Fichero hbm o fichero de correspondencia: Son ficheros propios de Hibernate en los que está especificada la correspondencia entre una clase y un esquema relacional (conjunto de tablas asociadas). Es decir, en un fichero de correspondencia se especifica cómo la información contenida en un objeto de una clase determinada se almacena en un esquema relacional.

HQL (Hibernate Query Language): Lenguaje parecido a SQL pero que maneja conjuntos de objetos con sus atributos en lugar de filas y columnas.

JPQL (Java Persistence Query Language): Es un subconjunto de HQL y es parte de la especificación de JPA.

Hibernate es un Framework que surge en el 2001 como una alternativa mejora a Java EE:

- Se trata de una solución que proporciona persistencia de objetos basada en ORM, pero sin necesidad de un servidor Java EE.
- Soporta transacciones. Utiliza técnicas de alto rendimiento, como el pool de conexiones, técnicas de agrupación por lotes (batching) para las bases de datos.

Hibernate.

Hibernate proporciona una API propia como una implementación de la de JPA. Hibernate 3 se convirtió en una versión certificada de JPA en el año 2010. Recordemos que JPA es la API estándar de Java para la persistencia de objetos y parte de la especificación para Java EE.

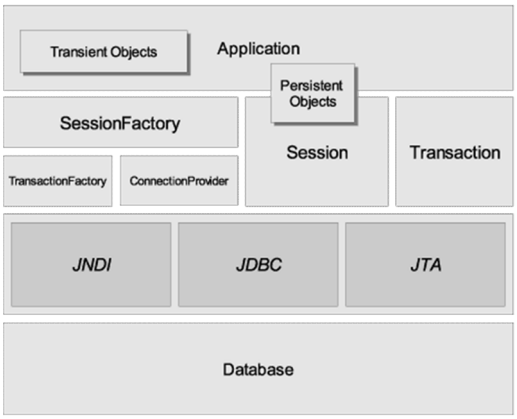
Hibernate tiene su propio lenguaje de consulta HQL para manejar objetos persistentes:

- Las sentencias SQL actúan sobre filas y columnas de las tablas.
- Las sentencias HQL actúan sobre un conjunto de objetos persistentes y sus atributos.
- JPQL: es el lenguaje estándar de JPA y es un subconjunto de HQL.

2. Arquitectura Hibernate

Las aplicaciones que usan Hibernate trabajan con **objetos persistentes** y **objetos transitorios**:

- Los objetos persistentes se asocian a una **session** creada por una **SessionFactory**. La aplicación puede crear objetos transitorios y convertirlos en persistentes para que puedan almacenarse en la BD.
- Las operaciones sobre objetos persistentes se pueden agrupar dentro de transacciones (**Transaction**) que están asociadas a una sesión.
- Las consultas **HQL** se pueden realizar mediante la clase **Query**.
- La interacción con la BD se hacer a través de **la api JPA o JDBC**. Aunque también puede utilizar internamente la API de Java JNDI para fuentes de datos DataSource y JTA para transacciones.



2. Arquitectura Hibernate

Las interfaces mostradas pueden clasificarse como sigue:

- Interfaces llamadas por la aplicación para realizar operaciones básicas (inserciones, borrados, consultas,...): *Session*, *Transaction*, y *Query*.
- Interfaces llamadas por el código de la infraestructura de la aplicación para configurar Hibernate. La más importante es la clase *Configuration*.
- Interfaces callback que permiten a la aplicación reaccionar ante determinados eventos que ocurren dentro de la aplicación, tales como *Interceptor*, *Lifecycle*, y *Validatable*.
- Interfaces que permiten extender las funcionalidades de mapeado de Hibernate, como por ejemplo *UserType*, *CompositeUserType*, e *IdentifierGenerator*.

Además, Hibernate hace uso de APIs de Java, tales como JDBC, JTA (**J**ava **T**ransaction **A**pi) y JNDI (**J**ava **N**aming **D**irectory **I**nterface).

2. Arquitectura Hibernate

- La 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.

- La 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.

- La 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*.

2. Arquitectura Hibernate: *SessionFactory*.

• 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.

• 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*).

Hibernate: la correspondencia objeto relacional.

Aunque la correspondencia entre objetos Java y datos de una BD relacional es compleja, nosotros empezaremos con el caso más simple, en el que, para cada clase hay una tabla en el modelo relacional que permite almacenar los objetos de esa clase.

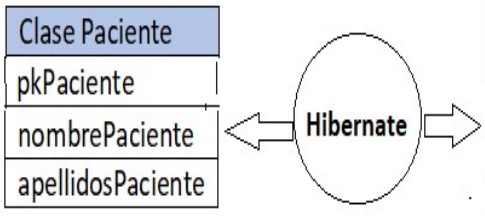


	Tabla paciente	
pkPaciente	nombrePaciente	apellidosPaciente
1	Javier	Abascal
2	Ana	Alonso
3	Oscar	Arregui

Hibernate: la correspondencia objeto relacional.

A partir de esta correspondencia simple se contemplan otras más complejas que deben representar:

- Colecciones de objetos para las relaciones de 1: N o de N:M en la BD.
- Referencias a objetos para las relaciones 1:1 o de N:1 en la BD.
- Herencia.

IMPORTANTE: Si ya tenemos una base de datos, Hibernate nos permite establecer una correspondencia Objeto Relacional. De esta forma, podemos aprovechar estructuras ya existentes para construir aplicaciones actuales.

Anotaciones en el mapeo.

A la hora de hacer el mapeo, vemos que las anotaciones que aparecen para las clases cuando mapean las tablas y las relaciones utilizan una notación específica:

Las anotaciones que hemos visto cuando en una clase Java que mapeada con una tabla son:

- @Entity Indica que la clase es una tabla en la base de datos
- @Table(name = “nombre_tabla”, catalog = “nombre_base_datos”) Indica el nombre de la tabla y la base de datos a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración)

Y para los atributos simples mapeados con los campos de la tabla correspondiente:

- @Id Indica que un atributo es la clave
- @GeneratedValue(strategy = GenerationType.IDENTITY) Indica que es un valor autonumérico (PRIMARY KEY en MySQL, por ejemplo)
- @Column(name = “nombre_columna”) Se utiliza para indicar el nombre de la columna en la tabla donde debe ser mapeado el atributo.

Anotaciones en el mapeo.

Cuando partimos de una Base de Datos conocida, debemos atender a las siguientes condiciones:

- Las tablas deben tener clave primaria.
- Los valores de las columnas que formen parte de la clave primaria no pueden cambiar.
- Las clases deben implementar una interfaz Serializable.
- Las clases deben tener los métodos getX y setX implementados.

También se mapean las relaciones, de forma que se puedan implementar o mantener una relación entre clases bidireccional:

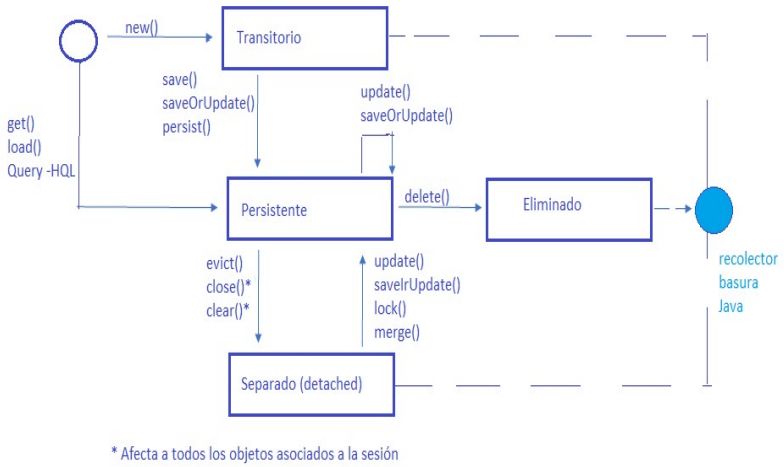
- @OneToOne Indica que el objeto es parte de una relación 1-1
- @ManyToOne Indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1
- @OneToMany Indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N
- @ManyToMany Indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos

Sesiones y estados de los objetos persistentes.

Hibernate nos ha permitido realizar la correspondencia ORM para clases de Java y para las relaciones entre ellas. Nos permite crear objetos transitorios que se han almacenado en la BD como objetos persistentes.

La clase para la que establecemos una correspondencia mediante Hibernate se llama clase persistente y podemos crear objetos persistentes instanciando la clase.

Ciclo de vida de los objetos persistentes:



Ciclo de vida de los objetos persistentes:

Este ciclo de vida muestra los estados en los que puede estar los objetos persistentes, las operaciones que permiten recuperarlos, modificarlos y grabar de nuevo estas modificaciones en la BD.

Una sesión (interfaz org.hibernate.Session) es esencial en Hibernate porque se construye sobre una conexión a la base de datos.

Las transacciones que hicimos en el ejemplo de clase son transacciones de sesión y pueden incluir varias operaciones sobre la BD. Una sesión puede mejorar el rendimiento de las consultas en objetos persistentes utilizando una caché. La apertura de sesiones se mejora usando un pool de conexiones.

Una sesión, junto con un gestor de entidades asociado, constituye un contexto de persistencia. El gestor de entidades (interfaz javax.persistence.EntityManager) lleva el control de los cambios que se realizan sobre objetos persistentes. La interfaz Session es de la API de Hibernate, mientras que la interfaz EntityManager pertenece a la de JPA. Es posible obtener una EntityManager a partir de una Session y al revés porque hay un puente entre ambas.

Los cambios que hacemos sobre los objetos quedan reflejados en la BD porque están asociados a un contexto de persistencia.

Los métodos save() y saveOrUpdate():

Pasan los objetos de una sesión de transitorios a persistentes: save(objeto).

La principal diferencia con saveOrUpdate(objeto) es que permite modificarlo.

Si tratamos de hacer persistente un objeto creado con new() y existe un objeto persistente de la misma clase e idéntico identificador, al hacer save(objeto) generar una excepción porque el identificador es único.

Si utilizamos saveOrUpdate(objeto) para el objeto creado con new(), funcionará como save(objeto). Pero si hay un objeto con el mismo identificador, lo modificará.

Los métodos close() y clear()

- El método session.clear() → separa todos los objetos asociados a una sesión, sin grabarlos.
- El método session.close() → graba todos los objetos asociados a la sesión y los separa.

Los métodos get() y load()

Sirven para obtener objetos persistentes.

Ambos métodos se usan para obtener un objeto persistente, recibiendo como argumentos la clase y el identificador único. La diferencia es:

- get(clase, identificador): devolverá null si no existe el objeto con ese identificador
- null(clase, identificador): Eleva ObjectNotFoundException.

El método delete(): Sirve para eliminar un objeto persistente: delete(objeto).

Los métodos update(), lock() y merge()

Estos métodos permiten pasar un objeto de separado a persistente.

- Método lock(). Útil cuando hay bloqueos.
- Método update() → Sirve para actualizar un objeto. También podemos usar saveOrUpdate().
- Método merge() → Sirve para actualizar los contenidos de otro objeto persistente, con el mismo identificador en la sesión si el objeto ya existe. En el caso de que no exista un objeto con ese identificador, lo crea.

Interfaz QUERY de Hibernate.

La API de Hibernate permite usar el lenguaje HQL para hacer consultas usando la interfaz QUERY que nos resultará útil para hacer consultas y cambios cuando tengamos que localizar objetos persistentes a partir de atributos o campos que no sean identificadores únicos.

La interfaz QUERY también es de JPA. El lenguaje de JPA se conoce como JPQL.

IMPORTANTE: Observa el manual de HQL facilitado en Moodle:

https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html

3. Configuración básica

Para utilizar Hibernate en una aplicación, es necesario conocer cómo configurarlo.
Hibernate puede configurarse y ejecutarse en la mayoría de aplicaciones Java y entornos de desarrollo.

Generalmente, Hibernate se utiliza en aplicaciones cliente/servidor de dos y tres capas, desplegándose Hibernate únicamente en el servidor.

Las aplicaciones cliente normalmente utilizan un navegador web, pero las aplicaciones *swing* y AWT también son usuales. Aunque solamente vamos a ver cómo configurar Hibernate en un entorno no gestionado, es importante comprender la diferencia entre la configuración de Hibernate para entornos gestionados y no gestionados:

3. Configuración básica

- **Entorno gestionado**: los *pools* de recursos tales como conexiones a la base de datos permiten establecer los límites de las transacciones y la seguridad se debe especificar de forma declarativa, es decir, en sus metadatos.

  Un servidor de aplicaciones J2EE, tal como JBoss, Bea WebLogic o IBM WebSphere implementan un entorno gestionado para Java.

- **Entorno no gestionado**: proporciona una gestión básica de la concurrencia a través de un *pooling* de *threads*. Un contenedor de servlets, como Tomcat proporciona un entorno de servidor no gestionado para aplicaciones web Java. Una aplicación *stand-alone* también se considera como no gestionada. Los entornos no gestionados no proporcionan infraestructura para transacciones automáticas, gestión de recursos, o seguridad. La propia aplicación es la que gestiona las conexiones con la base de datos y establece los límites de las transacciones.

Tanto en un entorno gestionado como en uno no gestionado, lo primero que debemos hacer es iniciar Hibernate. Para hacer esto debemos crear una *SessionFactory* desde una *Configuration*. A continuación explicamos cómo establecer las opciones de configuración de Hibernate.