IES CHAN DO MONTE
C.S. de Desenvolveento de Aplicacions Multiplataforma

UNIDAD 4: HIBERNATE –Configuración e operación con obxectos

Índice

1. Mapeo obxecto relacional

1.1 Desaxuste estrutural entre o modelo OO e o modelo relacional

A maioría das aplicacións que se utilizan están deseñadas para usar a ***Programación Orientadas a obxectos (POO)*** baseada **en *obxectos***. Os obxectos son entidades que teñen un determinado estado, comportamento (método) e identidade:

- O **estado** compóñese de datos ou información. Estes datos son representados por un ou varios atributos que almacenan valores concretos, o cal define a condición actual do obxecto. Exemplo

  Obxecto "Empregado":
  Id:1
  Nome: "María"
  Apelido: "García"
  Edade: 28 anos
  Departamento: "Finanzas"
  Cargo: "Analista Financeiro"
  Salario: 2500 euros

  O estado pode cambiar no transcurso do tempo debido a diferentes eventos ou accións como promocións, aumentos salariais, cambios de departamento ou calquera outra actualización.

- O **comportamento** está definido polos métodos ou mensaxes ás que sabe responder o devandito obxecto, é dicir, que operacións se poden realizar con el. Exemplos

  Método **ascender(int id ,String novoCargo):** Actualiza o cargo do empregado.
  Método **aumentarSalario(int id,double porcentaxeAumento**): Ajusta o salario do empregado.
  Método **cambiarDepartamento**(int id,String novoDepartamento): Modifica o departamento do empregado.

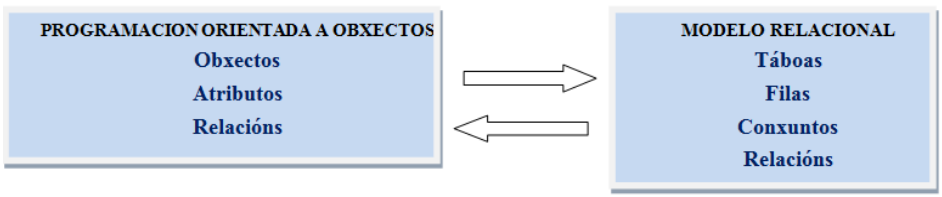
- A **identidade** é unha propiedade dun obxecto que o diferenza do resto.

  O atributo "**id" actúa como identificador único para cada obxecto "Empregado"**. Aínda que dous empregados poden compartir o mesmo nome ou apelido, o seu identificador único (id) será diferente, o que **lles proporciona a súa propia identidade dentro do sistema**.

O **estado dos obxectos débese persistir**, é dicir, preservalo de forma permanente (gardalo), para que, posteriormente, se poida recuperar a información deste (lelo) para a súa utilización de novo.

Para a persistencia dos obxectos existen varias técnicas. Unha das máis estendida é a ***utilización dunha base de datos relacional***.

Estas base de datos almacenan a información ***mediante táboas***, filas e columnas **e non se poden gardar de forma directa os obxectos nas táboas**.



Para *almacenar un obxecto* hai que realizar ***unha correlación entre o sistema orientado a obxectos da linguaxe e o sistema relacional*** da nosa base de datos.

As bases de datos relacionais **só poden gardar datos primitivos**, polo que non podemos gardar obxectos que vaiamos creando na nosa aplicación, senón que o que facemos ***é converter os datos do obxecto en datos primitivos para podelos almacenar nas táboas*** correspondentes das nosas bases de datos. Se logo necesitamos ese obxecto nalgunha parte da nosa aplicación, debemos de ***recuperar os datos primitivos da base de datos e volver construír o obxecto*.**

Outra diferenza está no establecemento das relacións. No modelo OO, as relacións entre clase represéntanse mediante referencias a obxectos ou coleccións de referencias a obxectos e poden ser bidireccionais. Estas referencias direccionais úsanse para navegar dende un obxecto a outro. No modelo relacional, as relacións represéntanse mediante claves foráneas (FK), e copias de valores en diferentes táboas. As claves foráneas non son inherentemente direccionais.

Outro problema é o que se refire á navegación. ***Traballando con obxectos***, saltamos dun obxecto a outro "paseando" (invocando métodos) polo **grafo das asociacións (referencias)**. Para *obter datos de* ***diferentes táboas*** nunha base de datos relacional implica ***realizar un ou máis (custosos) joins***.

Exemplo: En programación OO, para visualizar o nome do empregado que dirixe o departamento 1:

```
Departamento dept= (Departamento)sesion.get(Departamento.class,1);
String nome=dept.getEmpregado().getNomeCompleto();
```

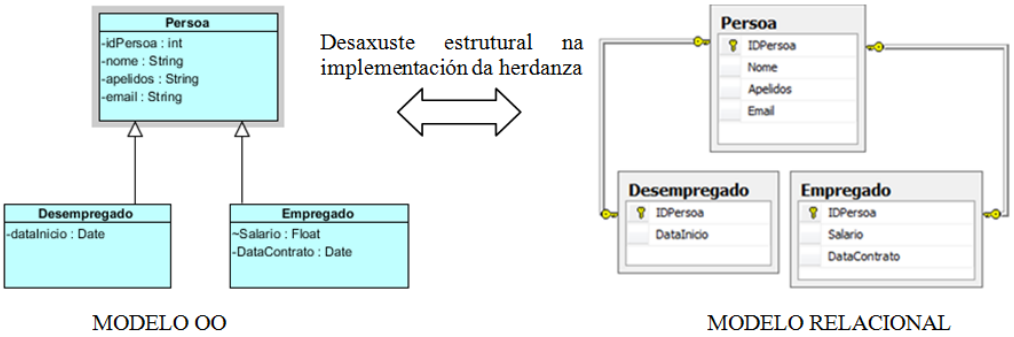
No modelo relacional, habería que facer unha consulta join:

```
SELECT Nome+' '+Apelido_1+' '+Apelido_2 FROM EMPREGADO INNER JOIN
DEPARTAMENTO ON NSS_dirixe=NSS
WHERE Num_departamento=1;
```

Ademais, *cando se invoca a linguaxe de consulta SQL dende unha linguaxe de programación* é necesario un mecanismo de vinculación que permita percorrer as filas dunha consulta á base de datos e acceder de forma individual a cada unha delas.

No modelo relacional non se pode modelar a herdanza que aparece no modelo orientado a obxectos. O paradigma OO dá soporte á herdanza, normalmente en forma de superclases e subclases e os obxectos herdan atributos e comportamentos dos seus obxectos pais, pola contra, os SXBDs non soportan herdanza en forma nativa e ,polo tanto, é necesario utilizar algunha estratexia para implementar a herdanza. As técnicas que se utilizan son: unha táboa por cada subclase, unha táboa por xerarquía ou unha táboa por clase.

Exemplo: Implementación da herdanza do modelo OO cunha táboa por clase no modelo relacional.



Existen tamén desaxustes nos tipos de datos, xa que os tipos de datos almacenados na base de datos difiren dos utilizados nas linguaxes de programación.

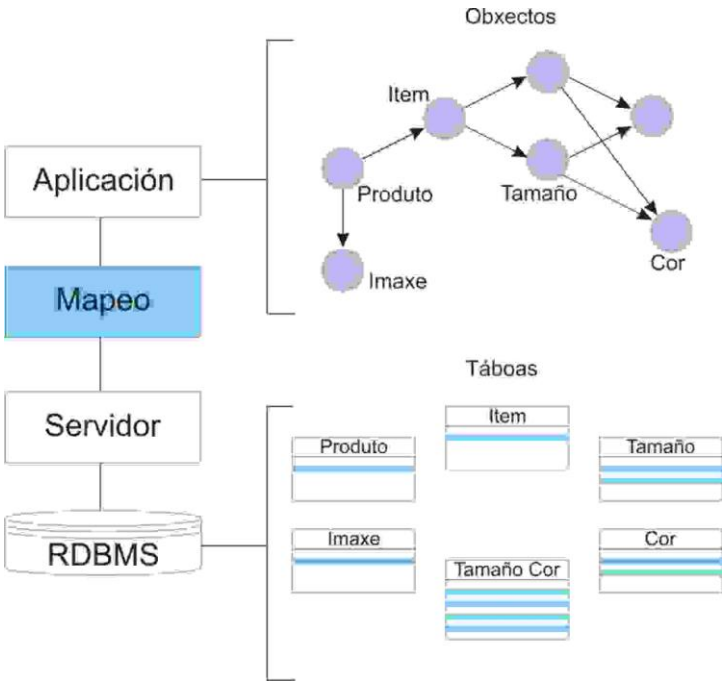
Tipos de datos JAVA	**Tipos de datos SQL**
boolean	En SQL, **non hai un tipo de datos boolean directo** en todas as bases de datos. Algúns sistemas de bases de datos, como PostgreSQL ou MySQL, fornecen tipos como BOOLEAN ou BOOL, pero outros sistemas poden usar tipos numéricos ou de cadea de texto para representar valores booleanos o bit (0,1) como Sql Server
String	En SQL, utilízase o tipo de datos **VARCHAR, CHAR** ou **TEXT** para almacenar cadeas de texto. A lonxitude máxima dunha cadea en SQL xeralmente especificase ao definir a columna (por exemplo, VARCHAR(255)).
int	O tipo de datos **INT** en SQL utilízase para almacenar números enteiros. Tamén hai variacións en tamaños, como **TINYINT, SMALLINT, INTEGER** ou **BIGINT**, que representan diferentes tamaños de enteros segundo a precisión necesaria.
double ou float	En SQL, utilizanse tipos como **FLOAT ou DOUBLE** para representar números en punto flotante. A precisión e o almacenamento destes tipos poden variar segundo a base de datos.
Date, Calendar,LocalDate, LocalDateTime,LocalTime, (para Java 8 e posteriores)	En SQL utilízase tipos con **DATE, TIME, DATETIME, TIMESTAMP.** Os detalles da representación e a precisión poden variar entre diferentes sistemas de bases de datos

Estes **desaxustes nos tipos de datos entre Java e SQL poden necesitar conversión ou xestión especial** ao interactuar entre a capa de aplicación en Java e a base de datos relacional. Empregan técnicas como mapeo obxecto-relacional (ORM) ou consultas parametrizadas para xestionar estas diferenzas e asegurar a coherencia no intercambio de datos entre a aplicación e a base de datos.

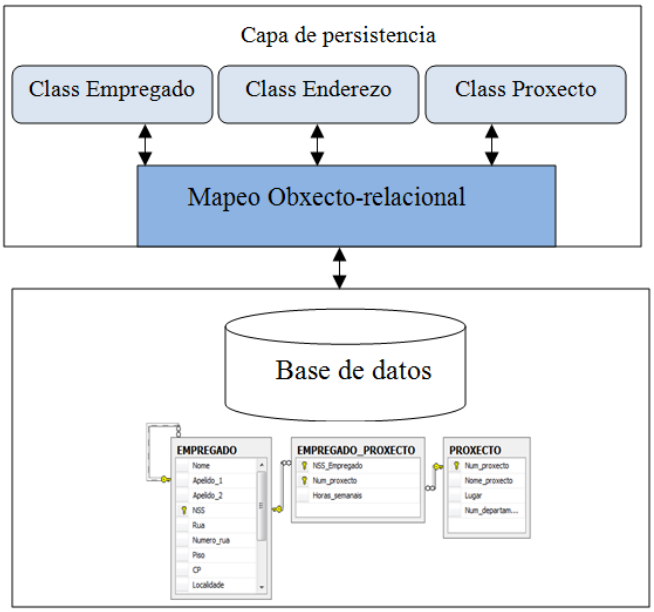
1.2 Concepto de mapeo obxecto relacional

Para *atenuar os efectos do desaxuste entre os dous modelos* existen varias técnicas e unha das máis usada é o *mapeo Obxecto/Relacional*.

ORM (Object Relational Mapping - Mapeo Relacional de Obxectos) *é unha técnica de programación para converter datos entre a linguaxe de programación orientada a obxectos e o sistema de xestión de base de datos relacional utilizado no desenvolvemento de aplicacións*



A importancia da utilización das técnicas de *mapeo obxecto_relacional* é que *converten, de forma automática, os obxectos nos datos primitivos almacenados nas táboas da base de datos relacional, simulando así o acceso a un sistema de base de datos orientados a obxectos*. Isto posibilita o uso de características propias da orientación a obxectos como herdanza e polimorfismo. Na práctica, o mapeo obxecto-relacional crea unha base de datos virtual orientada a obxectos sobre a base de datos relacional.



1.3 Ferramentas ORM. Vantaxes e inconvenientes

Entre as características dunha ferramenta ORM, podemos citar:

- **Permítenos traballar directamente coas clases** deseñadas no noso modelo de dominio, sen ter que traballar con filas das táboas da base de datos.
- **Permite traballar con diferentes base de datos** (SQLServer, Oracle, MySQL, Post-GreSQL...) e para cambiar dunha base de datos a outra só habería que realizar o cambio no ficheiro de configuración.
- **Xera automaticamente o código SQL necesario para o acceso á base de datos**, usando un mapeo obxecto relacional, que se especifica mediante un documento XML ou por anotacións.
- Permite **crear, modificar, recuperar e eliminar obxectos persistentes** e ademais permítenos navegar polas asociacións entre os obxectos e actualizalos ao final dunha transacción.

Vantaxes

- **Rapidez no desenvolvemento**. A maioría das ferramentas actuais permiten a creación do modelo por medio do esquema da base de datos, lendo o esquema, créanos o modelo axeitado.
- **Abstracción da base de datos**. Ao utilizar unha ferramenta ORM, evitamos a inclusión de sentenzas SQL embebidas no código da aplicación, separandonos totalmente do sistema de Base de datos que utilicemos. Isto facilita a migración cara a outro sistema xestor de bases de datos sendo o cambio máis sinxelo
- **Reutilización**. Permítenos utilizar os métodos dun obxecto de datos dende distintas zonas da aplicación, mesmo dende aplicacións distintas.
- **Seguridade**. Os ORM adoitan implementar sistemas para evitar tipos de ataques como poden ser os SQL injections.
- **Mantemento do código**. Facilítanos o mantemento do código debido á correcta ordenación da capa de datos, facendo que o mantemento do código sexa moito mais sinxelo.
- **Linguaxe propia para realizar as consultas**. Estes sistemas de mapeo traen a súa propia linguaxe para facer as consultas, o que fai que os usuarios deixen de utilizar as sentenzas SQL e pasen a utilizar a linguaxe propia de cada ferramenta.

Desvantaxes

- **Tempo utilizado na aprendizaxe**. Este tipo de ferramentas adoitan ser complexas polo que a súa correcta utilización leva un tempo que hai que empregar en ver o funcionamento correcto e ver todo o partido que se lle pode sacar.
- **Aplicacións algo mais lentas**. Isto é debido a que todas as consultas que se fagan sobre a base de datos, o sistema primeiro deberá transformalas á linguaxe propia da ferramenta ORM, despois ler os rexistros e por último crear os obxectos.
- **Posíbel pérdida de rendemento**: Aínda que as ferramentas ORM ofrecen flexibilidade e abstracción, ás veces poden xerar **consultas menos eficientes que as escritas manualmente**. Isto pode causar un rendemento inferior, especialmente en sistemas que manexan grandes cantidades de datos ou consultas moi específicas e complexas.
- **Dependencia da calidade do mapeamento**: A eficacia dunha ferramenta ORM depende moito da calidade do mapeamento definido entre os obxectos da aplicación e as táboas da

base de datos. Un **mapeamento incorrecto pode causar problemas de rendemento**, consultas ineficientes ou incluso perda de datos.

- **Actualizacións automáticas e operacións complexas**: En certas situacións, as actualizacións automáticas da base de datos poden ser máis complicadas de xestionar. Por exemplo, durante actualizacións masivas, o mecanismo automático dun ORM pode ser menos eficiente ou necesitar configuracións especiais para optimizar o proceso.
- **Costo de recursos**: As ferramentas ORM poden precisar máis recursos de hardware e memoria debido ás operacións adicionais para mapear obxectos a táboas e viceversa, o que pode resultar nun maior consumo de recursos en comparación con operacións directas coa base de datos.

Estas desventaxes non son universais e poden variar dependendo do contexto da aplicación e da forma en que se utiliza a ferramenta ORM en particular. É importante avaliar coidadosamente as necesidades do proxecto antes de decidir se a utilización dunha ferramenta ORM é a mellor opción.

Resumo:

O uso dunha ferramenta ORM, é beneficioso en moitos casos, especialmente **cando se traballa cunha estrutura de datos complexa ou cando se require portabilidade entre diferentes bases de datos**. Tamén é útil cando se busca acelerar o desenvolvemento, evitar a escrita manual de sentenzas SQL e **abstraerse da base de datos**.

Pode non ser axeitado en situacións **onde o rendemento é unha prioridade crítica ou cando se require unha optimización fina de consultas** para **volumes enormes de datos**. Ademais, para proxectos pequenos ou simples, o uso de Hibernate pode ser excesivo e engorxoso, resultando nun gasto de recursos innecesario.

Hibernate é vantaxoso para aplicaciones complexas, cando a abstracción da base de datos é esencial e cando a portabilidade e a velocidade no desenvolvemento son prioritarias. Non obstante, debe valorarse caso por caso para determinar se é a mellor elección, especialmente se a prioridade é o rendemento ou para proxectos máis pequenos e sinxelos onde a complexidade do ORM non se xustifique.

Na actualidade hai moitos tipos de framework que nos devolven o mapeo obxecto-relacional, segundo a linguaxe que esteamos a utilizar. Imos nomear algúns dos máis utilizados:

Hibernate

É unha das ferramentas **ORM máis utilizadas no mundo Java** e está dispoñible como **software libre baixo a licenza GNU LGPL** (Lesser General Public License).

Proporciona un marco de traballo que **simplifica o mapeamento de obxectos Java a táboas de bases de datos**. Utiliza **arquivos de configuración XML ou anotacións** para definir o mapeamento entre as clases Java e as táboas da base de datos. **Hibernate xera consultas SQL automaticamente**, facilitando a interacción entre as clases de Java e a base de datos relacional.

Hibernate **tamén ofrece unha opción de consulta chamada HQL**, que é semellante a SQL pero está **orientada a obxectos**. HQL permite aos desenvolvedores escribir consultas orientadas a obxectos e logo Hibernate as traduce a consultas SQL para interactuar coa base de datos. Isto proporciona un nivel adicional de abstracción e flexibilidade ao escribir consultas.

NHibernate é un framework similar e influenciado por Hibernate, pero *está deseñado específicamente para a plataforma .NET*. NHibernate segue moitos dos mesmos principios e conceptos de Hibernate, permitindo aos desenvolvedores traballar co mapeamento obxecto-relacional nun estilo semellante ao de Hibernate, pero na plataforma .NET.

Entity Framework (EF) Core:

É unha ferramenta ORM de Microsoft utilizada na plataforma .NET.

Permite aos desenvolvedores traballar con datos relacionais usando obxectos .NET e abstrae a interacción coa base de datos. Ofrece soporte para diferentes provedores de bases de datos e permite a creación de aplicacións de .NET que poden traballar con SQL Server, MySQL, PostgreSQL, entre outros.

Django ORM:

É un framework de **desenvolvemento web en Python**, e o seu ORM é unha parte integral do mesmo.

Django ORM facilita a creación, manipulación e consulta de datos relacionais utilizando obxectos Python. Proporciona unha abstracción sobre a base de datos relacional, o que permite aos desenvolvedores interactuar coa base de datos utilizando modelos de Python, simplificando así as operacións de base de datos.

Sequelize:

É unha **ferramenta ORM para Node.js**, compatible con varios dialectos de bases de datos SQL como MySQL, PostgreSQL, SQLite e MSSQL.

Ofrece unha forma fácil e flexible de traballar con bases de datos relacionais utilizando JavaScript. Sequelize permite definir modelos e relacións entre eles, e xera consultas SQL automaticamente.

Nota: Node.js é un entorno de execución de JavaScript baseado no motor V8 de Google Chrome. Á diferenza de outros entornos de JavaScript que se executan no lado do cliente (como nun navegador web), Node.js está deseñado para executarse no lado do servidor.

Doctrine

É un framework **ORM para PHP 5.2** e versións posteriores. Un dos seus puntos fortes é a súa linguaxe DQL (Doctrine Query Language), inspirada no HQL de Hibernate.

Ofrece funcionalidades avanzadas de mapeamento objeto-relacional e simplifica a interacción con bases de datos relacionais en aplicaciones PHP.

Propel

É outro framework **ORM para PHP 5 e superior**, amplamente utilizado **no marco do framework Symfony**, o cal é un completo framework deseñado para optimizar o **desenvolvemento de aplicacións web segundo o patrón Modelo Vista Controlador (MVC).** Propel facilita o acceso e manipulación de datos nunha base de datos relacional dende aplicativos PHP.

SQLAlchemy

É un toolkit de código aberto SQL e unha ferramenta ORM para a linguaxe de programación Python. Publicada baixo a licenza MIT, é unha das ferramentas máis utilizadas xunto con Django para acceder a bases de datos relacionais dende aplicaciones Python. Ofrece funcionalidades potentes para xerar consultas SQL e interactuar coa base de datos.

JPA(Java Persistence API

É **unha especificación de Java que describe un estándar para o mapeamento obxecto-relacional** en aplicativos Java. ***Non é propiamente un framework***, senón un **estándar definido** pola comunidade de desenvolvedores.

A pesar de non ser un framework en si mesmo, JPA define un **conxunto de interfaces e especificacións que as implementacións ORM** poden seguir para proporcionar unha capa de abstracción entre a base de datos relacional e as clases de obxectos Java.

Existen **varias implementacións de JPA**, sendo **Hibernate unha das máis coñecidas** e utilizadas. Hibernate, a pesar de ser unha implementación de JPA, ofrece funcionalidades adicionais máis alá do estándar JPA, o que o fai unha ferramenta poderosa e amplamente empregada no mundo Java.

Outras implementacións de JPA inclúen **EclipseLink**, **Apache OpenJPA**, **DataNucleus**, entre outras.

Estas implementacións seguen as especificacións definidas por JPA e ofrecen funcionalidades comúns para o mapeamento obxecto-relacional en aplicativos Java, permitindo aos desenvolvedores traballar con obxectos Java e interactuar coa base de datos de forma sinxela e estándar.

Nesta unidade imos traballar con Hibernate que é unha ferramenta ORM das máis utilizadas.

Hibernte e o estándar JPA (Java Persistence API)

Hibernate segue o estándar JPA (Java Persistence API) en varios aspectos importantes:

- Permite o uso **das anotacións JPA estándar** para o mapeamento obxecto-relacional.

  Estas anotacións, como **@Entity, @Table, @Id, @Column**, entre outras, usanse para definir entidades, relacións e atributos das entidades. Isto facilita a creación do modelo de datos e a definición da asignación entre as clases de Java e as táboas da base de datos.

- Proporciona unha **implementación de EntityManager**, que é a interface estándar de JPA utilizada para **realizar operacións CRUD** (Crear, Ler, Actualizar, Eliminar) nas entidades. Esta interface permite aos desenvolvedores realizar accións como persistir, buscar, actualizar e eliminar entidades, seguindo o estándar JPA.

- Permite o uso de **JPQL** (Java Persistence Query Language), que é a linguaxe de consulta estándar de JPA.

  JPQL é unha **linguaxe de consultas orientada a obxectos** que permite realizar consultas independentes do motor da base de datos específico, facilitando a portabilidade do código entre diferentes provedores de bases de datos.

  HQL (Hibernate Query Language) é o lenguaxe de consulta propio de Hibernate que se basa en JPQL en canto á sintaxe e funcionalidade. HQL é específico de Hibernate e úsase dentro do entorno de desenvolvemento de Hibernate para facer consultas en obxectos gardados.

- Segue **o ciclo de vida estándar das entidades** definido por JPA. As entidades pasan por diferentes estados como **transient, persistent, detached e removed**, e Hibernate xestiona estes estados segundo as operacións realizadas no contexto de persistencia.

- Admite **a xestión de transaccións** utilizando o estándar JPA. Permite a xestión de transaccións a través de EntityManager, o que inclúe o inicio, confirmación e reversión de transaccións.

Aínda que Hibernate ofrece funcionalidades adicionais máis alá do estándar JPA, segue a especificación JPA en moitos aspectos fundamentais. Isto significa que *as aplicacións desenvolvidas utilizando Hibernate poden ser compatibles con outras implementacións de JPA* e ofrecen certo grado de portabilidade entre diferentes provedores de persistencia JPA.

2. Hibernate

Hibernate é unha ferramenta ORM completa que conseguiu, nun tempo record, unha excelente reputación na comunidade de desenvolvemento, posicionándose, claramente, como un dos produtos OpenSource líder neste campo, grazas ás súas prestacións, boa documentación e estabilidade.

Empezouse a desenvolverse fai algúns anos por Gavin King, sendo hoxe Gavin e Christian Bauer os principais xestores do seu desenvolvemento.

Hibernate parte dunha filosofía de mapear obxectos Java "normais", coñecidos na comunidade como ***"POJOS"*** (Plain Old Java Objects). Para almacenar e recuperar estes obxectos, o programador debe manter unha conversación co motor de Hibernate, mediante un obxecto especial que é a sesión.

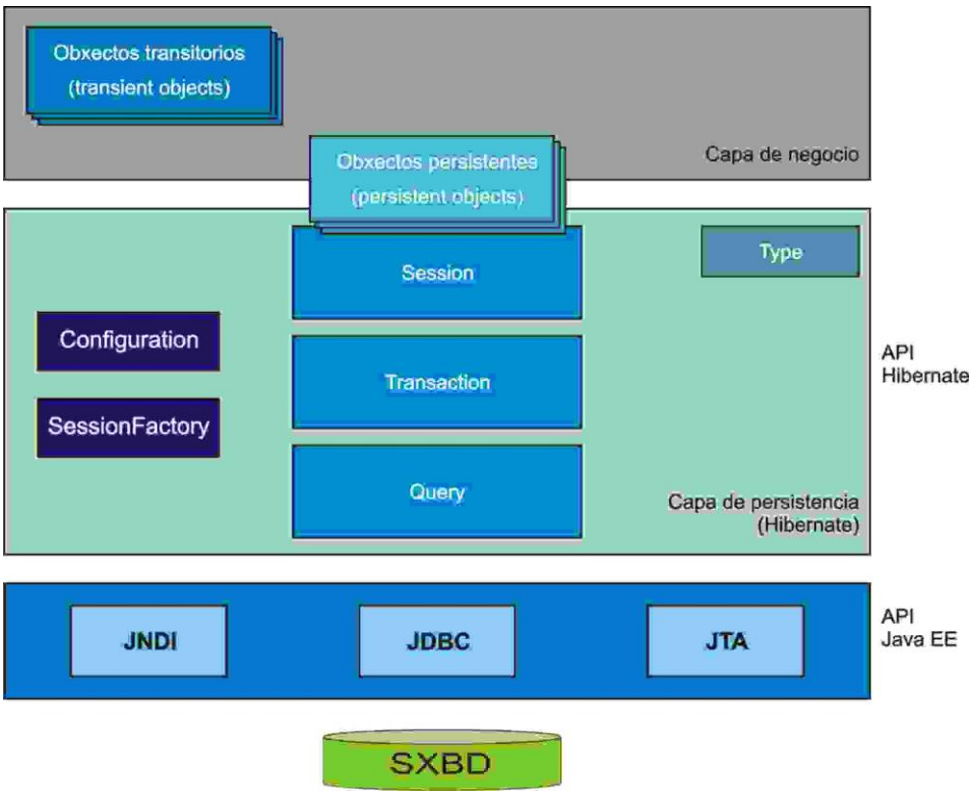
Nota: POJO é un acrónimo para *Plain Old Java Objects*. O nome utilízase para resaltar que, o obxecto en cuestión, é un obxecto Java normal, non é un obxecto especial, e non é en particular un Enterprise JavaBean (EJB3 -Non é unha soa clase, senón un modelo completo de compoñente). O termo foi acuñado por Martin Fowler, Rebecca Parsons e Josh MacKenzie, en setembro de 2000.

Un POJO é un obxecto serializable, ten un ***constructor sin argumentos***, e permite o *acceso ás propiedades* utilizando os métodos ***getter e setter*** .

2.1 Arquitectura Hibernate

O API de Hibernate é unha arquitectura de dúas capas (Capa de persistencia e Capa de dominio (ou negocio).

Na seguinte figura, móstranse as interfaces Hibernate máis importantes nas capas de persistencia e de negocio dunha aplicación Java. A capa de negocio está situada sobre a capa de persistencia, debido a que actúa como un cliente da capa de persistencia.



- **Capa de Persistencia**: Esta capa é onde Hibernate xoga un papel importante. **A través desta capa, Hibernate facilita a interacción coa base de datos**. É aquí onde os obxectos da nosa aplicación se mapean ás táboas da base de datos.

  Hibernate utiliza a ***API JDBC para interactuar directamente coa base de datos***, enviando consultas SQL e recuperando datos.

- **Capa de Dominio ou de Negocio**: Esta capa contén **a lóxica de negocio da aplicación**. É onde se definen as regras de negocio e os procesos que a aplicación debe executar.

  Estes obxectos representan as entidades e as súas relacións no sistema, como os usuarios, produtos, pedidos, etc. Aquí, os desenvolvedores definen as clases de obxectos e as súas relacións sen preocuparse directamente pola forma en que estes obxectos se almacenan na base de datos.

  Hibernate xoga un papel crucial para asegurarse de que os obxectos do dominio se mapeen correctamente á base de datos na Capa de persistencia.

Esta arquitectura de **dúas capas en Hibernate** permite **separar claramente as responsabilidades entre o traballo directo coa base de datos (Capa de persistencia) e a lóxica de negocio da aplicación (Capa de dominio),** facilitando así un desenvolvemento máis estruturado e mantíbel da aplicación.

Nota:

JNDI (Java Naming and Directory Interface): É unha API de Java utilizada para acceder a servizos de directorio e obxectos a través de nomes. Permite que os aplicativos localicen e accedan a recursos distribuídos, como ordeadores, servidores de correo, bases de datos, etc. A través de nomes lóxicos, JNDI facilita a xestión de recursos distribuídos e a súa conexión dende os aplicativos Java.

JTA (Java Transaction API): É unha API de Java que proporciona un modelo de programación estándar para controlar transaccións en aplicativos Java. Axuda a coordinar transaccións distribuídas a través de varios recursos ou bases de datos. Xunto coa API JDBC, JTA permite realizar operacións atómicas, consistentes, illadas e duradeiras (conocidas como propiedades ACID) en transaccións, asegurando a integridade dos datos e a súa consistencia a través de diferentes fontes de datos dentro dunha aplicación distribuída.

As **Interfaces mostradas clasifícanse da seguinte forma**:

Interfaces chamadas pola aplicación para realizar operacións básicas: (insercións, borrados, modificacións, consultas, etc.)

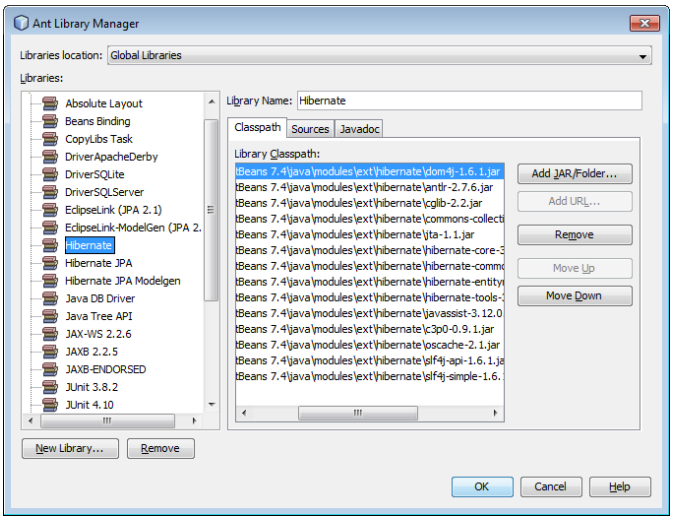
- ***Session*:** Interface primaria utilizada en calquera aplicación Hibernate para establecer unha conversación con Hibernate. A interface SessionFactory permite obter instancias Session.
- ***Transaction***: Permite controlar as transaccións.
- ***Query*:** Permite realizar peticións á base de datos e controla como se executa a devandita petición (query). As peticións escríbense **en HQL** ou no dialecto SQL nativo da base de datos que estamos a utilizar. Unha instancia Query utilízase para enlazar os parámetros da petición, limitar o número de resultados devoltos pola petición e para executar a devandita petición.

Interfaces chamadas polo código da infraestrutura da aplicación para configurar Hibernate.

A máis importante é a clase ***Configuration**:* utilízase para *configurar* e *arrancar Hibernate*. A aplicación utiliza unha instancia de Configuration para especificar a situación dos documentos que indican o mapeado dos obxectos e propiedades específicas de Hibernate, e a continuación, crea un obxecto **`SessionFactory`.**

Ademais, Hibernate fai uso de APIs de Java, tales como JDBC, JTA (Java Transaction Api) e JNDI (Java Naming Directory Interface -é usada por Java RMI e as APIs de Java EE para buscar obxectos nunha rede).

Hibernate **está dispoñible mediante unha serie de librarías `jar`**. Polo tanto, para poder realizar unha aplicación con Hibernate, temos que engadir estas librarías ao noso proxecto.



2.2 Como traballa Hibernate

- **Todas as consultas e operacións de inserción**, actualización e supresión de datos na base de datos da aplicación, ímolas a realizar **utilizando obxectos**. Todas estas interaccións realízanse nas instancias das clases mapeadas por Hibernate, é dicir, no *noso código non vai haber referencias directas ás táboas e columnas da base de datos*.
- **No corazón de cada interacción**, entre o código e a base de datos, atópase a **sesión de Hibernate**. A sesión de Hibernate é un **obxecto `Session`**, que encarna o concepto dun servizo de persistencia (ou xestor de persistencia) e serve de ponte durante unha conversación entre a aplicación e a base de datos. **Todas as operacións na base de datos, ímolas a realizar no contexto dunha sesion.**

  Imos ter métodos para gardar un obxecto -`save(obxecto)`-, para actualizalo -`update(obxecto)`-, para borralo -`delete(obxecto)`), etc., *sen necesidade de especificar unha sentenza SQL*.
- Cada **sesión envolve a conexión JDBC subxacente** e serve como un primeiro nivel caché para obxectos persistentes ligados a ela. A sesión é un obxecto moi lixeiro e non consome moita memoria, polo que a súa creación e destrución non custa moito. Isto é moi importante, xa que a nosa aplicación necesitará crear e destruír sesións todo o tempo. ***Para crear sesións, utilízase un obxecto SessionFactory.***
- Hibernate garda nos **arquivos de configuración `cfg.xml`, toda a información necesaria para conectarse a cada base de datos**

  Hai que **establecer a correspondencia entre as clases da aplicación e as táboas da base de datos** (como os campos e propiedades das clases Java se mapean nas columnas da base de datos): se puede hacer con **ficheros de mapeo (`hbm.xml`) o anotacións nas clases**.

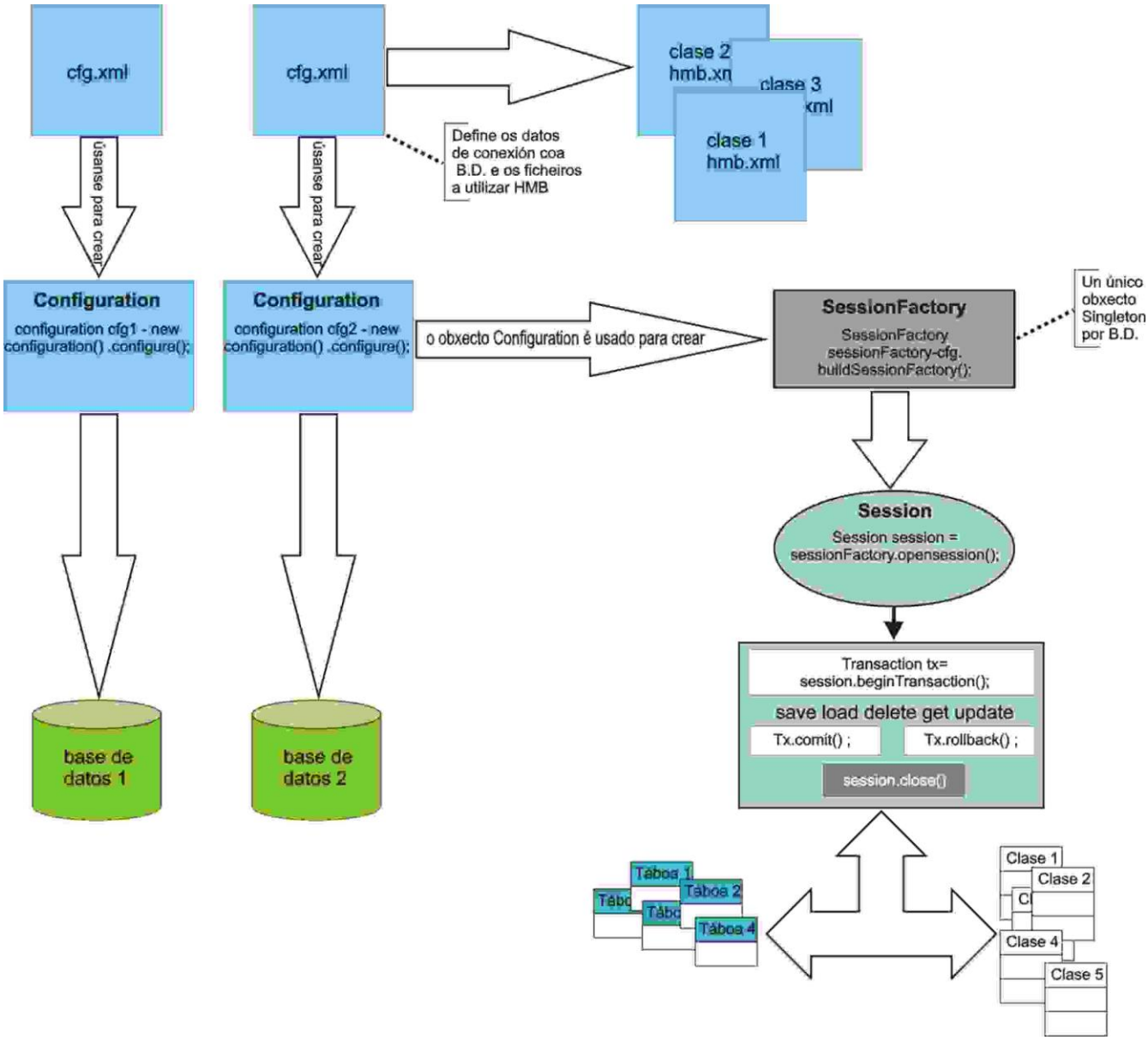
  No caso de utilizar **ficheros de mapeo (`hbm.xml`),** en estos se deben declarar a ubicacion deste ficheiros no ficheiro de configuración **`cfg.xml`**

  Tamén permite o uso de **anotacións para realizar este mapeamento** *sen a necesidade de arquivos de configuración XML separados*. As anotacións, como @Entity, @Table, @Column, entre outras, poden ser utilizadas directamente no código das clases Java.

O uso de anotacións en Hibernate permite definir o mapeamento directamente nas clases, o que pode simplificar a configuración, especialmente en aplicaciones con múltiples entidades, e facilita a lectura e comprensión do mapeamento ao ter todo o código relacionado nun único lugar.

Si Hibernate **accede a máis dunha base de datos, necesitarase un arquivo de configuración para cada unha delas**.

- Cada un destes arquivos específicos de configuración de cada base de datos, xunto cos ficheiros de mapeo (se o caso de mapeo), compílanse e almacénanse nunha caché polo obxecto `SessionFactory`. Para crear o obxecto `SessiónFactory`, utilízase a interface `Configuration`. Cada SessionFactory está configurada para traballar cunha determinada plataforma de base de datos.
- `SessionFactory` é un obxecto pesado que, idealmente, debe ser creado só unha vez (xa que é moi custoso de crear e implica unha operación lenta). O obxecto `SessionFactory` está posto a disposición do código da aplicación para realizar as operacións de persistencia previstas. Cada vez que se queira realizar estas operacións, créase un obxecto `Session` e realízanse baixo o contexto da sesión.

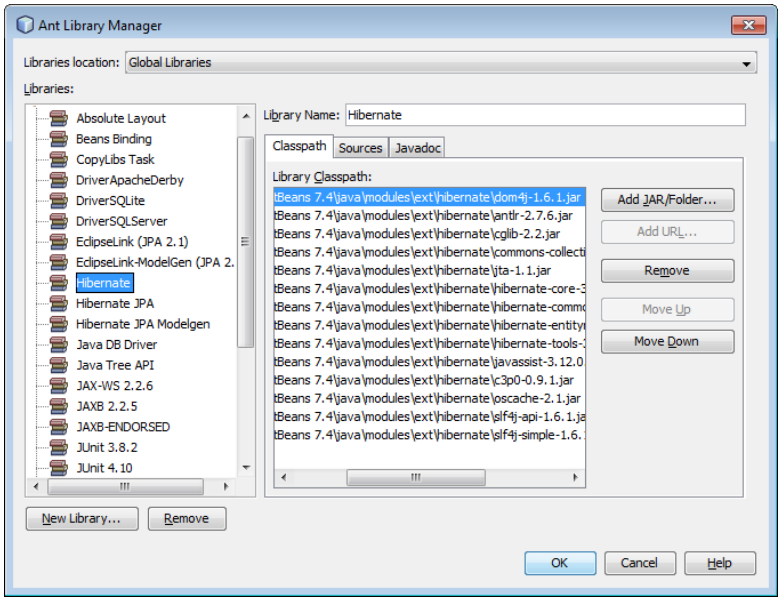


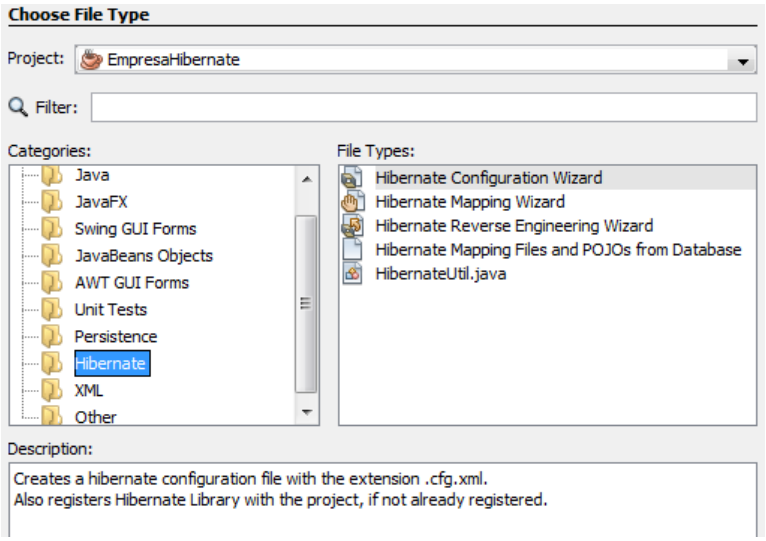
2.3 Descargar Hibernate

Ata a versión do IDE NetBeans 8.2, xa inclúe as bibliotecas de Hibernate preinstaladas e ven con asistentes ou ferramentas integradas que facilitan a creación e configuración do Hibernate. Estas ferramentas poden axudar na creación dos arquivos de configuración, mapeamento e clases necesarias para traballar con Hibernate.

Na versión do IDE NetBeans 12 con jdk13, pódese configurar asistentes ou ferramentas integradas para facilitar a creación e configuración de Hibernate mediante un plugin chamado "Hibernate Support"

Por exemplo, instalando o IDE Netbeans 7.4 temos dispoñible Hibernate 4.



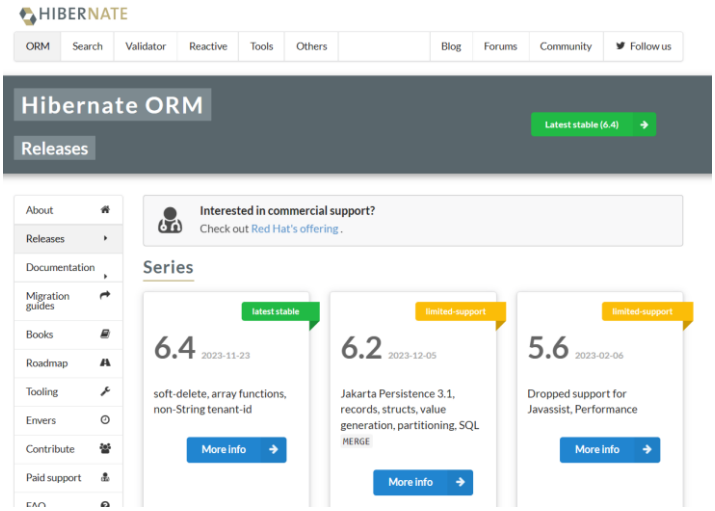


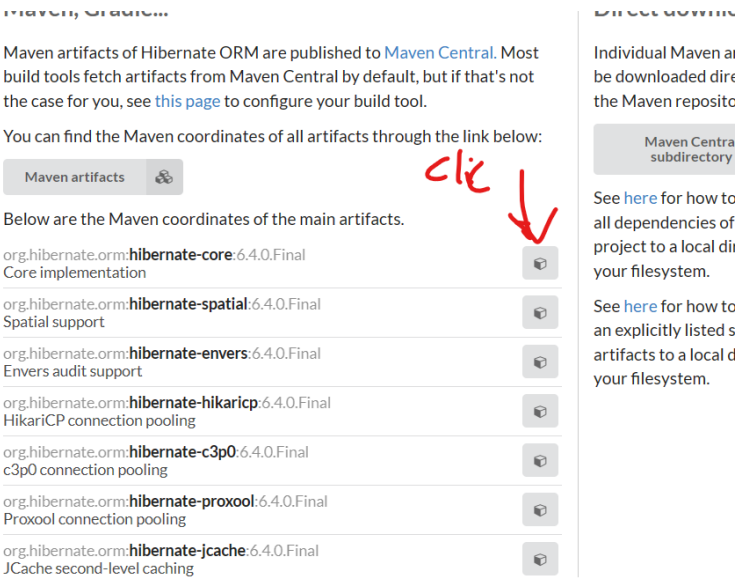
Se non dispoñemos das librarías de Hibernate no noso IDE de programación, o primeiro que temos que facer é descargalas. Podémolo facer do sitio de descarga oficial de Hibernate:

https://hibernate.org/orm/releases/

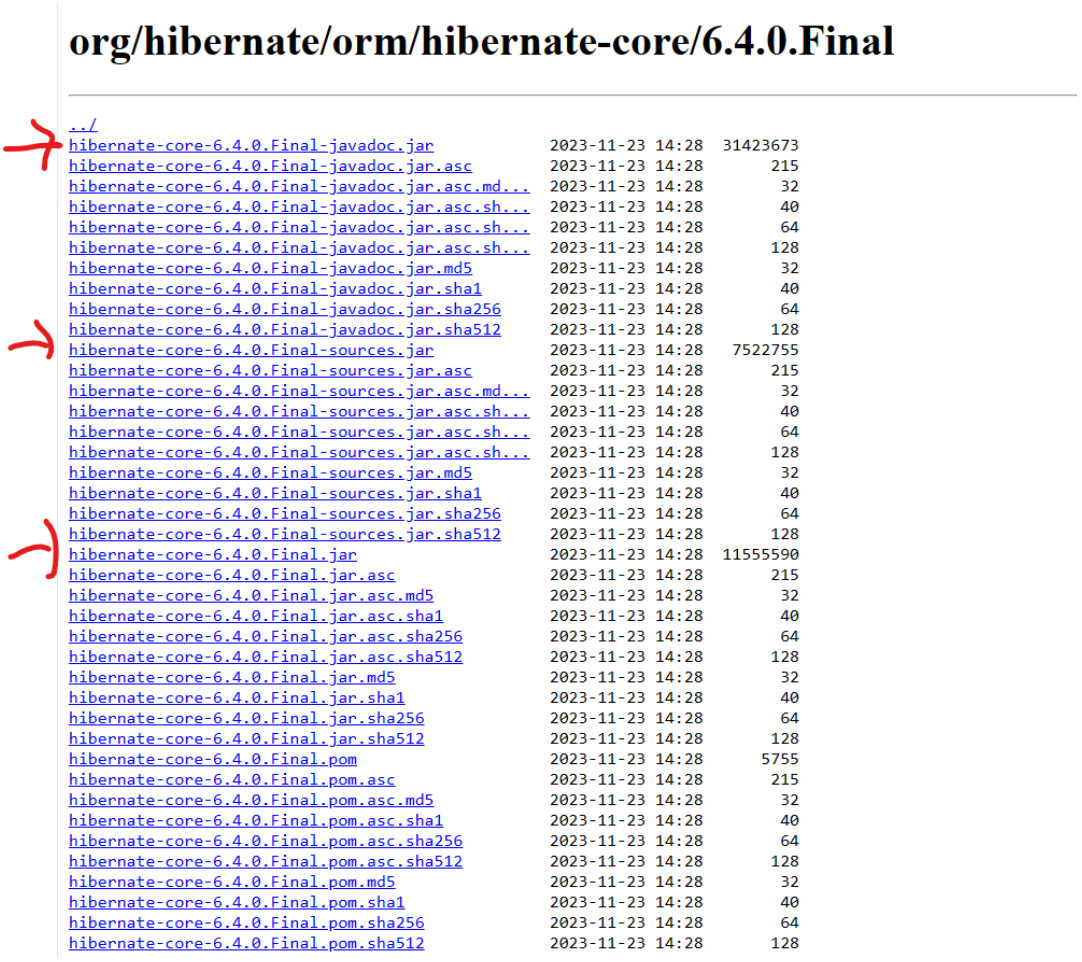
Descargamos a última versión estable dispoñible.

Neste caso, pódese descargar Hibernate 6.4 que a última.





More information about specific releases (announcements, download links) can be found here.



org/hibernate/orm/hibernate-core/6.4.0.Final

../		
hibernate-core-6.4.0.Final-javadoc.jar	2023-11-23 14:28	31423673
hibernate-core-6.4.0.Final-javadoc.jar.asc	2023-11-23 14:28	215
hibernate-core-6.4.0.Final-javadoc.jar.asc.md...	2023-11-23 14:28	32
hibernate-core-6.4.0.Final-javadoc.jar.asc.sh...	2023-11-23 14:28	40
hibernate-core-6.4.0.Final-javadoc.jar.asc.sh...	2023-11-23 14:28	64
hibernate-core-6.4.0.Final-javadoc.jar.asc.sh...	2023-11-23 14:28	128
hibernate-core-6.4.0.Final-javadoc.jar.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final-javadoc.jar.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final-javadoc.jar.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final-javadoc.jar.sha512	2023-11-23 14:28	128
hibernate-core-6.4.0.Final-sources.jar	2023-11-23 14:28	7522755
hibernate-core-6.4.0.Final-sources.jar.asc	2023-11-23 14:28	215
hibernate-core-6.4.0.Final-sources.jar.asc.md...	2023-11-23 14:28	32
hibernate-core-6.4.0.Final-sources.jar.asc.sh...	2023-11-23 14:28	40
hibernate-core-6.4.0.Final-sources.jar.asc.sh...	2023-11-23 14:28	64
hibernate-core-6.4.0.Final-sources.jar.asc.sh...	2023-11-23 14:28	128
hibernate-core-6.4.0.Final-sources.jar.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final-sources.jar.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final-sources.jar.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final-sources.jar.sha512	2023-11-23 14:28	128
hibernate-core-6.4.0.Final.jar	2023-11-23 14:28	11555590
hibernate-core-6.4.0.Final.jar.asc	2023-11-23 14:28	215
hibernate-core-6.4.0.Final.jar.asc.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final.jar.asc.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final.jar.asc.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final.jar.asc.sha512	2023-11-23 14:28	128
hibernate-core-6.4.0.Final.jar.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final.jar.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final.jar.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final.jar.sha512	2023-11-23 14:28	128
hibernate-core-6.4.0.Final.pom	2023-11-23 14:28	5755
hibernate-core-6.4.0.Final.pom.asc	2023-11-23 14:28	215
hibernate-core-6.4.0.Final.pom.asc.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final.pom.asc.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final.pom.asc.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final.pom.asc.sha512	2023-11-23 14:28	128
hibernate-core-6.4.0.Final.pom.md5	2023-11-23 14:28	32
hibernate-core-6.4.0.Final.pom.sha1	2023-11-23 14:28	40
hibernate-core-6.4.0.Final.pom.sha256	2023-11-23 14:28	64
hibernate-core-6.4.0.Final.pom.sha512	2023-11-23 14:28	128

Para facer funcionar Hibernate, os arquivos máis importantes son os seguintes:

- `hibernate-core-6.4.0.Final.jar`: Este é o arquivo JAR principal que contén as clases e bibliotecas necesarias para executar Hibernate.
- `hibernate-core-6.4.0.Final-sources.jar`: Este arquivo contén o código fonte de Hibernate, que pode ser útil para a depuración e o entendemento do funcionamento interno de Hibernate.
- `hibernate-core-6.4.0.Final-javadoc.jar`: Este arquivo contén a documentación de JavaDoc para Hibernate, que pode ser útil para entender o propósito e uso das clases e métodos individuais en Hibernate.

Para utilizar hibernate, engádese o arquivo hibernate-core-6.4.0.Final.jar ás bibliotecas do teu proxecto, tamén necesitarase engadir o controlador JDBC correspondente á base de datos. Este controlador é o que permite a Hibernate comunicarse coa a base de datos.

Por exemplo, se estás a usar MySQL como a túa base de datos, necesitarás o arquivo mysql-connector-java.jar. Se estás a usar PostgreSQL, necesitarás o arquivo postgresql.jar.

Unha vez que engadas estes arquivos JAR ás bibliotecas do teu proxecto, deberías poder usar Hibernate para mapear as túas clases de Java ás táboas da túa base de datos e realizar operacións de base de datos.

2.4 Configuración

Debido a que Hibernate foi deseñado para que se poida traballar en distintos ámbitos, existen unha gran cantidade de parámetros de configuración. Hibernate é un sistema altamente configurable para adaptarse a isto.

Para poder traballar con Hibernate, débese realizar:

- A **configuración do servizo Hibernate**.
- **Proporcionar a Hibernate toda a información asociada ás clases** que se queren facer persistentes é un paso crucial no uso de Hibernate. Isto pode facerse mediante o uso de ficheiros de mapeo ou anotacións nas clases.
  - **Os ficheiros de mapeo** son ficheiros XML que definen como se mapean as clases de Java ás táboas da base de datos. Cada clase persistente ten un ficheiro de mapeo correspondente que especifica as súas propiedades e como se mapean a columnas específicas na táboa da base de datos.
  - **As anotacións** son unha alternativa aos ficheiros de mapeo. Proporcionan a mesma información que os ficheiros de mapeo, pero están integradas directamente no código da clase. Isto pode facer que o código sexa máis fácil de entender e manter, xa que a información de mapeo está localizada xunto coa clase que afecta.

2.4.1 Configuración do servizo Hibernate

Hibernate proporciona dous posibles ficheiros de configuración:

- *Ficheiro XML* normalmente chamado `hibernate.cfg.xml`.
- *Ficheiro de propiedades* estándar de Java (`hibernate.properties`).

Ambos ficheiros, levan a cabo a mesma tarefa (configuración do servizo Hibernate). Si ambos ficheiros se atopan no classpath da aplicación, o primeiro sobrescribe ao segundo.

Estes ficheiros utilízanse para configurar o tipo de conexión que se vai xerar contra a base de datos e as clases que se van asociar ás táboas.

`Ficheiro hibernate.cfg.xml.`

Este arquivo é lido por Hibernate cando arranca. A estrutura xeral dun ficheiro XML de configuración é algo como isto:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory name="nome sesión">
        <!-- parámetros de configuración -->
        <property name="propiedade">valor</property>
        <!-- especificación dos parámetros de configuración -->

        <!-- especificación dos arquivos de mapeo -->
        <mapping resource="Ficheiro de mapeamento *.hbm.xml"/>
        <!-- ou -->
        <!-- especificación as entidades coas anotacions (@Entity) -->
        <mapping class="clases entidades"/>
        <!-- ..... -->
    </session-factory>
</hibernate-configuration>
```

A continuación, amósase un exemplo de configuración `hibernate.cfg.xml` utilizando fichero de mapeo `hbm.xml`:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Configuración da base de datos -->
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/Agenda</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">abc123</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>

        <!-- Mapeo de clases -->
        <mapping resource="Mapeo/Persona.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

A continuación, amósase un exemplo de configuración `hibernate.cfg.xml` utilizando anotacións:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Configuración da base de datos -->
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/Agenda</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">abc123</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>

        <!-- Mapeo de clases -->
        <mapping class="Pojos.Persona"/>
    </session-factory>
</hibernate-configuration>
```

Aquí observamos a gran importancia do ficheiro de configuración, pois é aquí **onde se especifica que base de datos usamos**, polo que si cambiásemos de base de datos, abondaría con cambiar este ficheiro de configuración, mantendo nosa aplicación intacta.

A estrutura do ficheiro `hibernate.cfg.xml` ven marcada por un ficheiro de validación `hibernate-configuration-3.0.dtd`.

- Contén os datos de conexión coa base de datos
- Sinala a situación dos ficheiros de mapeo das clases persistentes.

A etiqueta `<SessionFactory>` consta das etiquetas `<Property>` e `<Mapping resource>`. Si se dispón de varias bases de datos, hai que ter un SessionFactory por cada unha delas.

- **Nas propiedades (Property)** defínense os elementos necesarios para *establecer as conexións*. Todos os nomes das propiedades que soporta Hibernate, están definidos en `org.hibernate.cfg.Environment`
- **Na etiqueta Mapping** sinálase *onde está o ficheiro XML co mapeo entre a clase java e a táboa* da base de datos enlazada o los pojos coas anotacións.

Algunhas destas propiedades son:

✓ **Datos da conexión: datos para conectar coa Base de datos.**

- **hibernate.connection.driver_class**: Esta propiedade especifica o controlador JDBC que se usará para conectar coa base de datos. Neste caso, estás a usar o controlador JDBC de MySQL, que é com.mysql.jdbc.Driver.
- **hibernate.connection.url**: Esta propiedade especifica a URL da túa base de datos. Neste caso, a URL é jdbc:mysql://localhost:3306/Agenda, o que significa que estás a conectar a unha base de datos MySQL chamada Agenda no teu servidor local.
- **hibernate.connection.username** e **hibernate.connection.password**: Estas propiedades especifican o nome de usuario e o contrasinal que se usarán para conectar coa base de datos. Neste caso, o nome de usuario é root e o contrasinal é abc123.
- **hibernate.dialect**: Esta propiedade especifica o dialecto de SQL que Hibernate debe usar, o que nos permite xerar un SQL optimizado para unha base de datos relacional en particular. O dialecto é unha especie de "tradutor" que permite que Hibernate xere SQL que é compatible co teu sistema de xestión de bases de datos. Neste caso, estás a usar o dialecto de MySQL 5, que é org.hibernate.dialect.MySQL5Dialect.Dialecto hibernate: hibernate.dialect

Si non se especifica, na maioría dos casos, Hibernate poderá seleccionar a implementación org.hibernate.dialect.Dialect correcta, en base aos metadatos que o controlador JDBC retorna.

RDBMS	Dialecto
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL5	org.hibernate.dialect.MySQL5Dialect
MySQL5 with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL con MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (cualquier versión)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseASE15Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
H2 Database	org.hibernate.dialect.H2Dialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Imaxe da páxina https://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html_single/#configuration-optional-dialects

- ✓ **Propiedades para habilitar información de log (mellor deshabilitalo nun ámbito en produción) .**

  hibernate.show_sql-> Habilita mostrar por consola todas as sentenzas SQL executadas por Hibernate. Valores: false ou true. O valor por defecto para esta propiedade é false, o que significa que Hibernate non mostrará as consultas SQL que xera.

  hibernate.format_sql -> Imprime o SQL no rexistro e na consola. Valores: false ou true. O valor por defecto para esta propiedade é false, o que significa que Hibernate non formatará o SQL xerado para que sexa máis lexible.

  hibernate.use_sql_comments -> Hibernate engade comentarios a todas as sentenzas SQL xeradas para explicar a súa orixe. Valores: false ou true. O valor por defecto para esta propiedade é false, o que significa que Hibernate non engadirá comentarios ao SQL xerado.

- ✓ **Propiedades para a xestión de transaccións: Estas propiedades axudan a controlar o comportamento das transaccións en Hibernate.**

  hibernate.connection.autocommit: Esta propiedade especifica se as transaccións deben confirmarse automaticamente. Valores: false ou true. O valor por defecto para esta propiedade é false, o que significa que as transaccións non se confirman automaticamente.

- ✓ **Propiedades para a xestión de esquemas: Estas propiedades axudan a Hibernate a interactuar co esquema da base de datos.**

  hibernate.hbm2ddl.auto: Esta propiedade permite controlar o comportamento do esquema da base de datos, coas operacion crear, eliminar, actualizar ou validar as táboas baseándose nas túas clases de entidade. Os valores posibles son create, create-drop, update e validate. Non hai un valor por defecto para esta propiedade.

  - create: Se utiliza durante el desarrollo inicial, cuando estás creando tus entidades y quieres que Hibernate genere las tablas correspondientes. Create-drop borra ao finalizar as tábos
  - update: Se utiliza una vez que las entidades están bastante estables y quieres que Hibernate realice cambios incrementales en la base de datos a medida que modificas tus entidades.
  - validate: Se utiliza en producción, donde no quieres que Hibernate realice cambios en la base de datos. En su lugar, Hibernate comprobará que las entidades y las tablas coinciden.

  Si no se especifica la propiedad hibernate.hbm2ddl.auto, Hibernate no realizará ninguna operación automática en la base de datos2. Esto significa que se tendrá que gestionar el esquema de la base de datos manualmente.

- ✓ **Propiedades para a xestión de rendemento: Estas propiedades axudan a mellorar o rendemento de Hibernate.**

  hibernate.jdbc.batch_size: Esta propiedade especifica o número de actualizacións que Hibernate agrupará xuntas antes de enviar ao servidor da base de datos

Ficheiro Hibernate.properties

Os ficheiros `.properties` son simples ficheiros de texto que se adoitan utilizar para gardar parámetros de configuración, en forma de pares `clave-valor`.

Exemplo dun ficheiro de configuración de propiedades:

```
# Ficheiro de configuración Hibernate.properties
hibernate.connection.driver_class=com.microsoft.sqlserver.jdbc.SQLServerDriver
hibernate.connection.url=jdbc:sqlserver://;databaseName=Persoas
hibernate.connection.username=sa
hibernate.connection.password=abc123.
```

No Anexo explícase como crear o ficheiro `.properties` e os métodos principais para traballar con este ficheiro.

A continuación, mostrase algunhas páxinas onde podes aprender sobre as propiedades de configuración de Hibernate para establecer conexións JDBC:

1. [Hibernate - Configuration - Online Tutorials Library]: Este tutorial proporciona unha visión xeral das propiedades de configuración de Hibernate, incluíndo `hibernate.connection.driver_class`, `hibernate.connection.url`, e `hibernate.dialect`.
2. [Chapitre 3. Configuration]: Este documento de JBoss.org proporciona exemplos de configuracións de Hibernate, incluíndo a configuración de conexións JDBC.
3. [Hibernate Tutorial: Manage Hibernate Configuration File with ... - Oracle]: Este tutorial de Oracle explica como xestionar o ficheiro de configuración de Hibernate usando o editor de configuracións de Hibernate.
4. [how to configure hibernate config file for sql server] e [An example hibernate.cfg.xml for MySQL 8 and Hibernate 5]: Estas discusións de StackOverflow e TheServerSide proporcionan exemplos de ficheiros de configuración de Hibernate para SQL Server e MySQL, respectivamente.

3. Inicio de Hibernate

3.1 Creación dunha instancia SessionFactory

O primeiro paso para traballar con Hibernate é crear un `SessionFactory`. É necesario un `SessionFactory` por cada base de datos coa que se vaia traballar.

A interface `SessionFactory` representa á base de datos e proporciona instancias de obxectos `Session`. Estas *son compartidas por toda a aplicación*, en cambio as *instancias de sesión deben ser usadas só por unha única transacción* ou unidade de traballo.

Primeiro temos que crear un `obxecto Configuration` para indicarlle a Hibernate onde está o arquivo de configuración. Isto podémolo facer con:

```
Configuration cfg=new Configuration();
```

Métodos para indicarlle cal é o arquivo de configuración:

Configuration	**configure**() Usa o archivo de configuración: hibernate.cfg.xml.
Configuration	configure(Document document) Usa os mapeos e as propiedades especificados no obxecto document XML dado.
Configuration	configure(File configFile) Usa os mapeos e as propiedades especificados no obxecto File dado.
Configuration	configure(String resource) Use os mapeos e as propiedades especificados no String dado.
Configuration	configure(URL url) Usa os mapeos e as propiedades especificados na URL dada.

Exemplo para indicarlle o arquivo de configuración `hibernate.cfg.xml`. Este arquivo debe estar no *raíz da aplicación*.

```
Configuration cfg=new Configuration().configure();
```

A continuación debemos crear un obxecto `SessionFactory`. Para isto utilízase o método `buildSessionFactory()` da clase `Configuration`. A sintase deste método é:

SessionFactory	**buildSessionFactory**() **throws HibernateException** Crea un obxecto **SessionFactory** usando as propiedades da conexión e os arquivos de mapeos na configuración. Lanza: HibernateException – si se indica unha incorrecta información de configuración ou mapeo.

Exemplo de creación dun obxecto `SessionFactory`:

```
Configuration cfg=new Configuration().configure();
SessionFactory sessionFactory=cfg.buildSessionFactory();
```

A interface `SessionFactory` é compartida por toda a aplicación e vainos proporcionar instancias de obxectos `Session` (*obxectos para comunicarnos con Hibernate*), entón é recomendable ter unha clase especializada para crear as sesións.

Un bo hábito de desenvolvemento é construír unha clase chamada `HibernateUtil` que se apoie no patrón `Singleton`. O obxectivo de empregar este patrón é o *de garantir que unha clase só teña unha instancia na aplicación*, e proporcionar un único punto de acceso global a ela. Hibernate traballa con sesións e xera unha instancia da base de datos para traballar con ela, por tanto é lóxico é un bo hábito de traballo crear esta clase que proporciona un único punto de acceso á devandita instancia, evitando crear unha sesión distinta dende distintos puntos da aplicación que finalmente nos pode levar a xeración de erros e consumo innecesario de memoria.

O patrón Singleton garante que unha clase só teña unha instancia e proporciona un punto de acceso global a esta instancia. Este patrón adóitase utilizar cando unha clase controla o acceso a un recurso físico único ou cando hai datos que deben estar dispoñibles para todos os obxectos da aplicación e varios clientes distintos precisan referenciar a un mesmo elemento e queremos asegurarnos de que non hai máis dunha instancia dese elemento.

Para iso, desenvólvese a clase `HibernateUtil` na cal declaramos un atributo `static` do tipo `SessionFactory`, asegurándonos de que só existe unha instancia. Amais este atributo defínese como `final` para que non poida ser modificado nin alterado por ningún cliente que o referencie. Grazas ao patrón Singleton obtemos un acceso controlado da sesión, as clases que desexen unha referencia á sesión única obterana chamando ao método estático `getSessionFactory()` da clase. Exemplo:

```java
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

/**
 * Hibernate Utility class with a convenient method to get Session Factory
 * object.
 *
 * @author usuario
 */
public class HibernateUtil {

    // `SessionFactory` es una interfaz de Hibernate para crear sesiones
    private static final SessionFactory sessionFactory;

    // `StandardServiceRegistry` es una interfaz para mantener un registro c
    private static StandardServiceRegistry serviceRegistry;

    // Bloque estático para inicializar las variables estáticas
    static {
        try {
            // `Configuration` se utiliza para configurar Hibernate
            Configuration configuration = new Configuration().configure();

            // `StandardServiceRegistryBuilder` aplica configuraciones y cor
            serviceRegistry = new StandardServiceRegistryBuilder()
                .applySettings(configuration.getProperties()) // Aplicar prc
                .build(); // Construir el registro de servicios

            // Construir la `SessionFactory` usando la configuración y el re
            sessionFactory = configuration.buildSessionFactory(serviceRegist
        } catch (Throwable ex) {
            // Registrar la excepción en caso de error
            System.err.println("Fallo al crear la conexión" + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    // Método público para obtener la instancia de `SessionFactory`
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

- **SessionFactory**: Interfaz de Hibernate para la creación de sesiones de base de datos.
- **StandardServiceRegistry**: Interfaz para mantener un registro de servicios en Hibernate.
- **StandardServiceRegistryBuilder**: Clase para construir el registro de servicios aplicando configuraciones.
- **Configuration**: Clase para cargar y configurar las propiedades de Hibernate desde un archivo.

Clase HibernateUtil: Esta clase proporciona **una manera conveniente de obtener una instancia de SessionFactory**.

Bloque estático para inicialización: Este bloque se ejecuta una vez cuando la clase se carga en la JVM.

- configuration: Carga la configuración de Hibernate desde el archivo hibernate.cfg.xml.
- serviceRegistry: Aplica las configuraciones cargadas y construye el registro de servicios.
- sessionFactory: Construye la instancia de SessionFactory usando la configuración y el registro de servicios.
- Manejo de excepciones para registrar cualquier fallo durante la creación de la SessionFactory.

Método getSessionFactory():

- Método estático que devuelve la instancia de SessionFactory.
- Proporciona un punto de acceso global para obtener la instancia de SessionFactory.

3.2 Creación de obxectos Session

A responsabilidade principal dunha `SessionFactory` de Hibernate é crear e xestionar obxectos `Session`.

A responsabilidade principal dunha `Session` de Hibernate é proporcionar unha interface *CRUD* para as clases asignadas.

Nota: O acrónimo CRUD refírese ás 4 funcións principais que se executan nas aplicacións de bases de datos relacionais para o almacenamento persistente. Cada letra asígnase a cada unha das operacións: `Create (crear-INSERIR)`, `Read(ler, SELECT)`, `Update( Cambiar, UPDATE)` e `Delete (eliminar-Delete)`

Os obxectos Session proporcionan tres formas de facer consultas:

- Utilizando a linguaxe HQL (Hibernate Query Language). Esta linguaxe é a linguaxe de consulta orientada a obxectos de Hibernate. Permite realizar consultas similares a SQL, pero en termos de clases persistentes e as súas propiedades.
- Mediante a API de Programación para QBC (Query By Criteria). Dentro de Hibernate podemos encontrar a interface Criteria, que nos permite especificar consultas sobre as clases persistentes definindo un conxunto de restricións. QBC é unha alternativa a HQL, pero a diferenza dela, non é unha linguaxe de consulta, senón unha API de programación.
- Utilizando sentenzas SQL nativas para expresar consultas de base de datos. Só debe de utilizarse cando as outras opcións non son válidas (por exemplo, cando se necesite utilizar unha característica propia do SQL nativo da base de datos).

Para abrir unha conexión (sesión) utilízase o seguinte método da clase `SessionFactory`.

Session	**openSession** () Crea e abre unha conexión de base de datos

Exemplo:

```
public static void main(String[] args) {
        Session sesion=HibernateUtil.getSessionFactory().openSession();
  }
```

A `Session` proporciona unha caché (`contexto de persistencia`) onde se almacenan os estados dos obxectos cos que se van manexar na sesión, obxectos que proveñen da base de datos ou se almacenarán nela. Isto evita interaccións innecesarias coa base de datos. Cando executamos operacións como: `find()`, `update()`, `save(),saveOrUpdate()`, `get()`, `delete()` ou calquera outra operación da interface `Session`, estamos a interactuar de xeito transparente coa caché de Hibernate.

As operacións non se realizan directamente sobre a base de datos, almacénanse na caché.

Cada vez que abrimos unha sesión, obtense unha conexión a base de datos mediante o pool de conexións de Hibernate. Esta conexión libérase cando se pecha a sesión co método `close()`.

Cantas máis sesións manteñamos abertas, máis conexións estaremos consumindo do pool podendo chegar a esgotalo. Para solucionar isto, Hibernate permite desconectar unha sesión momentaneamente utilizando o método `disconnect()`. Esta operación, libera a conexión coa base de datos, aumentando efectivamente a escalabilidade do noso sistema ao evitar o esgotamento do pool de conexións. Para volver obter outra conexión a base de datos mediante o pool de conexións, podese utilizar o método `reconnect()`.

4. Traballando con obxectos

4.1 Comezando unha unidade de traballo

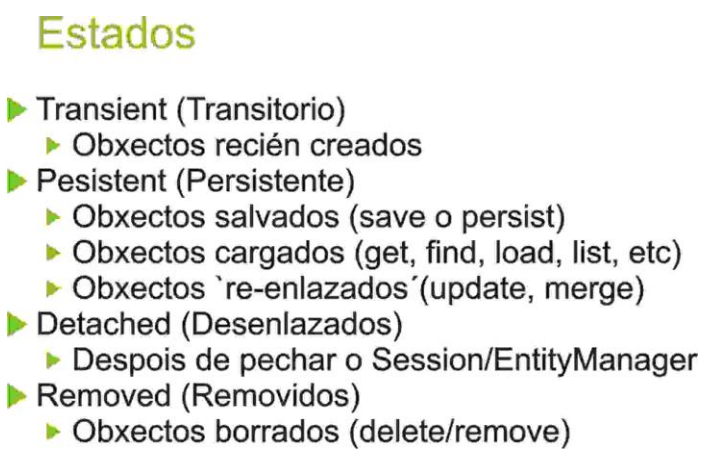
Para comezar unha unidade de traballo, execútanse as seguintes instrucións:

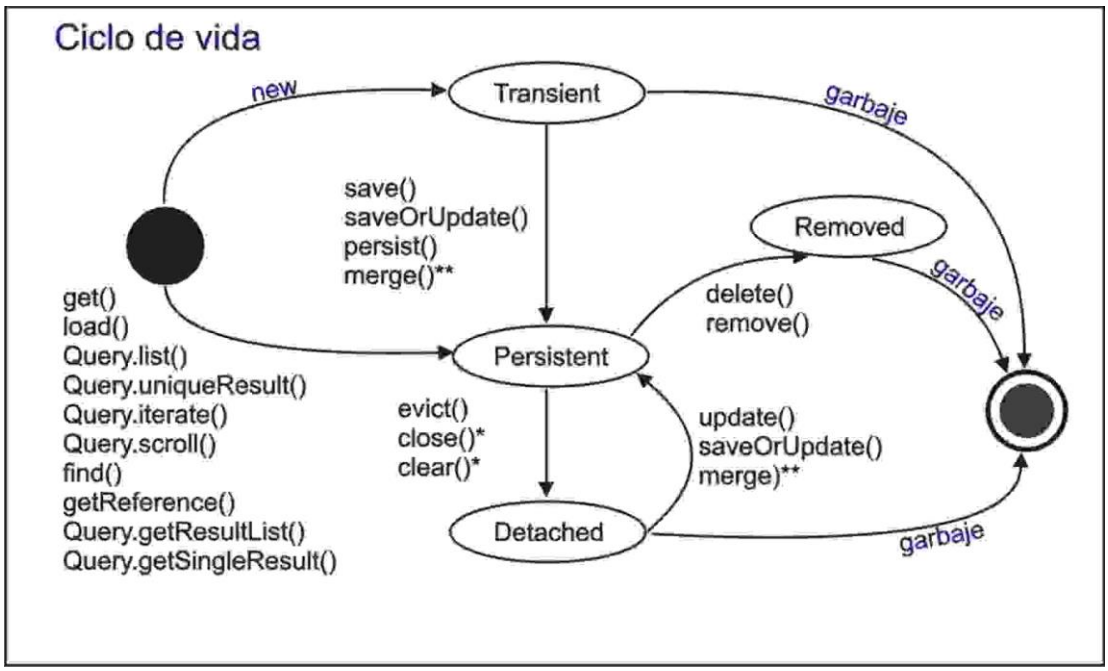
```
Session sesion = sessionFactory.openSession();
Transaction tx = sesion.beginTransaction();
```

- Neste punto, o *contexto de persistencia* é inicializado.
- A aplicación pode ter varios obxectos `SessionFactory`, cada un conectado a unha Base de Datos. A creación de SessionFactory é custosa, co que se aconsella inicializar o SessionFactory no arranque da aplicación unha única vez.
- A obtención dun obxecto `Session` é moi lixeira, de feito unha sesión non obtén a conexión JDBC ata que non é necesario, igual que a súa destrucción. Isto é importante, xa que a nosa aplicación necesitará crear e destruír sesións todo o tempo, quizás en cada petición.
- Na última liña ábrese unha transacción, e todas as operacións que se executen dentro da unidade de traballo realizaranse na mesma transacción.
- Proporciona un único fío que determina a conversación entre a aplicación e a base de datos nunha unidade atómica de traballo.

4.2 Ciclo de vida de obxectos persistentes

Os obxectos persistentes van pasar por varios estados manexados por Hibernate, facendo este proceso transparente. Hibernate define 4 estados, ocultando a complexidade da súa implementación ao usuario.





Transitorio (temporais)

Un obxecto é transitorio se **foi recen instanciado utilizando o operador `new`, e non está asociado a unha sesión de Hibernate**. Estes obxectos non son transaccionais, non teñen unha representación persistente na base de datos, non se lle asignou un valor identificador, non están asociados a ningún rexistro da táboa na base de datos, polo que o seu estado pérdese tan pronto como deixan de estar referenciados por algún outro obxecto. Ao perder a referencia a un obxecto transitorio, este será destruído polo colleiteiro de lixo.

Utilízase a sesión de Hibernate para facer un obxecto persistente. Hibernate ocúpase das declaracións SQL que necesitan executarse para esta transición.

Persistente

Un obxecto con estado persistente ten unha representación na base de datos e un valor identificador. Un obxecto pasa a estado persistente porque se garda, se obtén da execución dunha consulta ou porque se navegue polo grafo de asociacións dun obxecto persistente.

Os obxectos persistentes encóntranse no ámbito dunha sesión. Están asociados a un obxecto `Session` e son transaccionais. O seu estado é actualizado na BD ao final da transacción.

Por defecto escríbense todos os campos dunha táboa nunha actualización, pero pódese indicar que só se escriban as columnas modificadas, poñendo `'dynamic-update=true'` no ficheiro de mapeo no elemento `<class>`.

Hibernate detectará calquera cambio realizado a un obxecto en estado persistente e sincronizará o estado coa base de datos cando se complete a unidade de traballo.

Desconectados (separados ou desenlazados)

Unha instancia separada é un obxecto que se fixo persistente, pero a súa sesión foi pechada.

Para entender os obxectos desconectados, débese considerar unha transición típica dunha instancia: Primeiro é temporal, dado que acaba de ser creado na aplicación. Posteriormente vólvese persistente, ao chamar a unha operación do administrador de persistencia (a sesión). Todo isto

acontece dentro dunha soa unidade de traballo, e o contexto de persistencia para esta unidade de traballo está sincronizado coa base de datos nalgún punto.

A unidade de traballo complétase e o contexto de persistencia péchase. Non obstante, a aplicación aínda ten unha referencia á instancia que foi gardada. Sempre e cando o contexto de persistencia estea activo, o estado desta instancia é persistente, pero ao pecharse o seu estado cámbiase a desconectado.

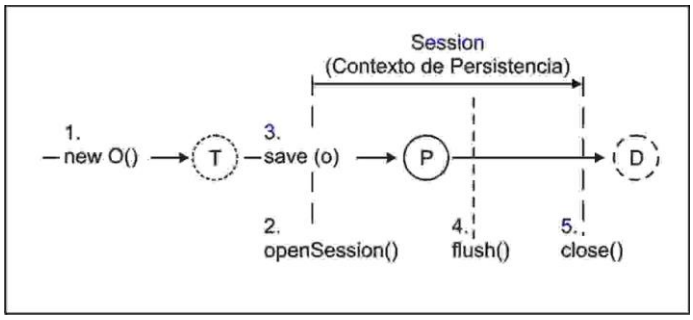
Os obxectos desconectados indican que o seu estado non garante que estean sincronizados co estado da base de datos, non obstante é posible seguir empregándoos e modificalos.

Obxectos borrados (removidos)

Un obxecto en estado borrado é planificado a ser borrado unha vez finalice a unidade de traballo (transacción). Por tanto, deberíase descartar calquera referencia a este obxecto.

4.3 Persistindo un obxecto

1. Instánciase un obxecto novo (estado `transient`).
2. Obtense unha sesión e comézase a transacción, *inicializando o contexto de persistencia.*
3. Unha vez obtida a sesión, chámase ao método `save()`, que introduce o obxecto no contexto de persistencia. Este método devolve o identificador do obxecto persistido.
4. Para que *os cambios sexan sincronizados na base de datos*, é necesario realizar o `commit()` da transacción ou dentro do obxecto sesión, chamar ao método `flush()`.Neste momento, obtense a conexión JDBC á base de datos para poder executar a oportuna sentenza.
5. Finalmente, a *sesión péchase*, libérase o contexto de persistencia, e xa que logo, a referencia do obxecto creado devólvese ao estado `detached`.



Exemplo:

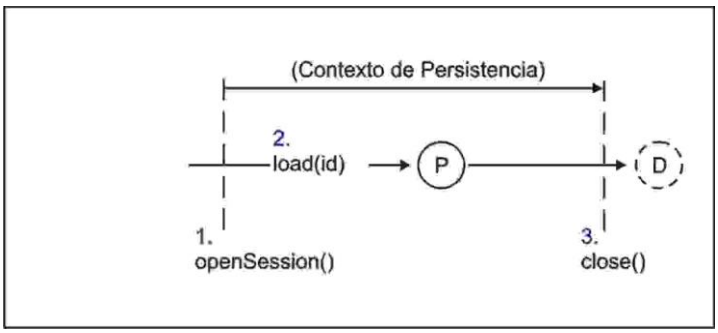
```
Session sesion=HibernateUtil.getSessionFactory().openSession();
Transaction tr=sesion.beginTransaction();
Persoa pers;
pers = new Persoa(4,"Lucia","Martin","Martin","lucia@gmail.es","76767676");
sesion.save(pers);
tr.commit();
sesion.close
```

4.4 Recuperando un obxecto persistente por id

Existen dous métodos que se encargan de recuperar un *obxecto persistente por identificador*: `load()` e `get()`.

O obxecto recuperado queda automaticamente ligado á sesión. A diferenza entre eles radica na forma de indicar que un obxecto non se encontra na base de datos:

- get() devolve un nulo.
- load() lanza unha excepción ObjectNotFoundException.



Por exemplo para recuperar o obxecto da táboa Persoa a través do seu identificador:

```
public static void main(String[] args) {
       Session sesion=HibernateUtil.getSessionFactory().openSession();
       Persoa per=(Persoa) sesion.get(Persoa.class,1);
       System.out.println("Nome:"+per.getNome());
       sesion.close();
    }
```

4.5 Modificando un obxecto persistente

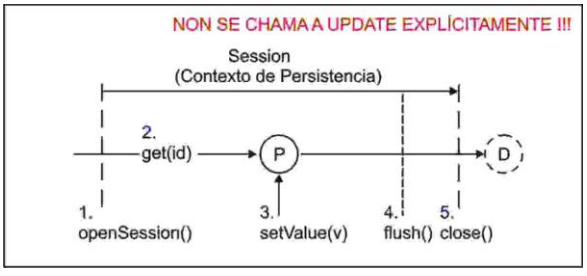
Para actualizar un obxecto invócase ao método `update` do obxecto Session. Exemplo:

```
public static void main(String[] args) {
       Session sesion=HibernateUtil.getSessionFactory().openSession();
       Transaction tr=sesion.beginTransaction();
       Persoa pers=(Persoa) sesion.get(Persoa.class,4);
       pers.setTelefono("777777777");
       sesion.update(pers);  //modifica o obxetco na base de datos
        tr.commit();
        pers=(Persoa) sesion.get(Persoa.class,4);
       System.out.println("Nome:"+pers.getNome()+"Telefono:"+pers.getTelefono());
```

Como dixemos que os obxectos persistentes se atopan no ámbito dunha Session, e Hibernate detectará calquera cambio realizado a un obxecto en estado persistente, e sincronizarao coa base de datos ao completar a unidade de traballo (cando se atope commit ou se chame ao método flush), entón no caso anterior, non fai falta especificar explicitamente o método update.

Exemplo:

```
public static void main(String[] args) {
       Session sesion=HibernateUtil.getSessionFactory().openSession();
       Transaction tr=sesion.beginTransaction();
       Persoa pers=(Persoa) sesion.get(Persoa.class,4);
       pers.setTelefono("777777777");
       tr.commit();
       pers=(Persoa)
       sesion.get(Persoa.class,4);
      System.out.println("Nome:"+pers.getNome()+" Telefono:"+pers.getTelefono());
```



4.6 Borrando un obxecto persistente

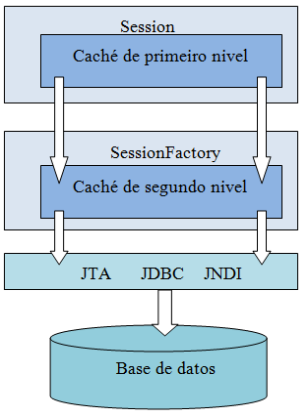
Para poder borrar un obxecto é necesario obtelo previamente.

Para borrar un obxecto invócase ao método `delete` do obxecto `Session`. Exemplo:

```
public static void main(String[] args) {
        Session sesion=HibernateUtil.getSessionFactory().openSession();
        Transaction tr=sesion.beginTransaction();
        Persoa pers=(Persoa) sesion.get(Persoa.class,34);
        sesion.delete(pers); //borra o obxecto da base de datos
        tr.commit();
```

4.7 Arquitectura da caché en Hibernate

Hibernate divide a estrutura da caché en dous niveis: A caché de primeiro nivel e a caché de segundo nivel.



4.7.1 Caché de primeiro nivel (Caché do contexto de persistencia)

A *caché do primeiro nivel* está sempre asociada ao obxecto `Session`, non pode desactivarse e non necesita configuración algunha.

Cada sesión aberta terá a súa propia caché de primeiro nivel proporcionando así *un contexto de persistencia*. Nesta caché vanse almacenando os obxectos que se recuperan da base de datos, de maneira que, si volven solicitarse, xa están na caché e non se fará unha nova consulta á base de datos. Na nosa aplicación podemos abrir tantas sesións como necesitemos e cada unha terá a súa propia cache de primeiro nivel.

Esta caché de primeiro nivel é o punto de acceso aos obxectos. Cando se realizan operacións de consulta e actualización ofrecidas pola interface `Session` interactúase de forma transparente coa caché de primeiro nivel. As operacións de actualización non se executan directamente, almacénanse primeiro na caché de primeiro nivel e para que o cambio sexa persistido é necesario realizar unha chamada ao método `flush()` da interface `Session` ou ao método `commit()` da transacción aberta para a sesión.

Como a caché de primeiro nivel é o contexto de persistencia dos obxectos en estado persistentes, pode ocurrir que se realice un uso excesivo desta caché chegando a provocar unha falta de memoria, por exemplo, nas operacións masivas de actualización ou no caso de que as asociacións entre os obxectos sexan non preguiceiras (neste caso, pode encherse a caché con elementos innecesarios).

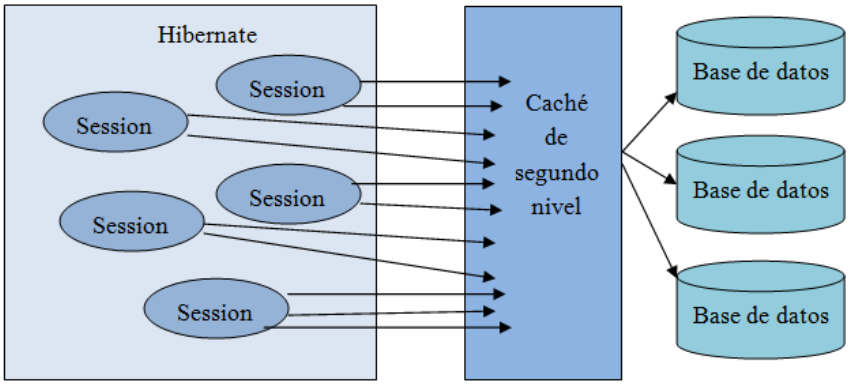
Podemos eliminar obxectos da caché, chamando:

- Ao método `evict(object)` que elimina o obxecto pasado como parámetro da caché. Este obxecto pasa a estado `detached`.
- Ao método `session.clear()` que elimina todos os obxectos que se encontran aloxados na caché, pasando todos estes obxectos a estado `detached`.

4.7.2 Caché de segundo nivel

A caché de segundo nivel permite mellorar o rendimento e o acceso concorrente por varios usuarios aos datos da base de datos. Esta caché evita os problemas que poden ocorrir nas operacións de actualización de datos concorrentes realizados en diferentes sesións.

Esta caché de segundo nivel está asociada co obxecto `SessionFactory` e trabállase con todos os obxectos recuperados e manexados por todas as sesións, la podríamos considerar como unha caché global. Agora, cando un obxecto non se encontra na caché de primeiro nivel (a sesión) Hibernate buscarao na caché de segundo nivel. Isto permítenos que todo obxecto pesistente dunha sesión poida ser obtido, as veces que sexan necesarias,en calquera lugar da aplicación e por calquera usuario.



Para que funcione a caché do segundo nivel, temos que habilitala. Para elo, facemos o seguinte:

- Seleccionamos un provedor de caché.

  Hibernate soporta 4 provedores de caché opensource:
  - EhCache (Easy Hibernate Cache) e a clase é org.hibernate.cache.EhCacheProvider
  - OSCache (Open Symphony Cache) e a clase é org.hibernate.cache.OSCacheProvider
  - Swarm Cache e a clase é org.hibernate.cache.SwarmCacheProvider
  - JBoss Tree Cache e a clase é org.hibernate.cache.TreeCacheProvider

Cada un destes provedores ten diferentes características no que respecta o rendemento, uso de memoria e configuración.

- Agregamos no ficheiro `hibernate.cfg.xml` as seguintes propiedades:

```
<property name="hibernate.cache.provider_class">
     Clase do provedor da caché
</property>
<property name="hibernate.cache.use_structured_entries">true</property>
```

- Indicamos no classpath, o arquivo de configuración xml do provedor dá caché (consultar instrucións do provedor da caché).
- Para cada clase que se queira utilizar a caché de segundo nivel para persistir os seus obxectos, agregamos a seguinte entrada:

```
<class name=" ">
   <cache usage="estratexia de  concorrencia"/>
   ....
</class>
```

- Estratexias de concorrencia

Resulta necesario establecer unha estratexia de concorrencia que permita sincronizar a caché de primeiro nivel coa caché de segundo nivel e esta última coa base de datos. Existen catro estratexias de concorrencia predefinidas. A continuación aparecen listadas por orde de restricións en termos do nivel de illamento transaccional.

- transactional: Garante un nivel de illamento ata lecturas repetitibles (repeatable read), se se necesita. É o nivel máis estrito. É conveniente o seu uso cando non poidamos permitirnos datos que queden desfasados. Esta estratexia só se pode utilizar en clusters, é dicir, con caches distribuídas.
- read-write: Mantén un illamento ata o nivel de commited, utilizando un sistema de marcas de tempo (timestamps). A súa maior utilidade dáse no mesmo caso que para a estratexia transactional pero coa diferenza de en que esta estratexia non se pode utilizar en clusters.
- nonstrict read-write: Non ofrece ningunha garantía de consistencia entre a caché e a base de datos. Para sincronizar os obxectos da caché coa base de datos utilízanse timeouts, de modo que cando caduca o timeout se recargan os datos. Con esta estratexia, temos un intervalo no cal temos o risco de obter obxectos desfasados. Cando Hibernate realiza unha operación flush() nunha sesión, invalídanse os obxectos da caché de segundo nivel. Esta operación é asíncrona e non temos nunca garantía de que outro usuario non poida ler datos erróneos. A pesar de todo isto, esta estratexia é ideal para almacenar datos que non sexan demasiado críticos.
- read-only: É a estratexia de concorrencia menos estrita. Ideal para datos que nunca cambian.

4.8 Transaccións

Un obxecto `Transaction` representa unha transacción física realizada contra a base de datos.

A transacción iníciase explicitamente cun beginTransaction, e péchase cun `commit` ou un `rollback`.

- Cando a transacción se confirma (commit), o estado dos obxectos persistentes sincronízase coa base de datos.
- Si facemos un rollback, Hibernate desfai todos os cambios realizados ata ese momento na base de datos, dentro da transacción.
  - Os cambios en memoria non se desfán.
  - Recordemos: unha vez recuperados os obxectos desde a base de datos, manter a súa consistencia en memoria é tarefa nosa, non de Hibernate.

É convinte pechar unha transacción dentro dun bloque `try catch finally`.

No control de concorrencia, Hibernate usa directamente conexións JDBC e recursos JTA sen agregar ningún comportamento de bloqueo adicional.

Hibernate non bloquea obxectos na memoria. A súa aplicación pode esperar ao comportamento definido polo nivel de illamento das súas transaccións no motor da base de datos.

Unha SessionFactory é un obxecto custoso de crear, pensado para que todos os fios da aplicación o compartan. Créase unha soa vez, usualmente no inicio da aplicación, a partir dunha instancia Configuration.

Unha Session é un obxecto de baixo custo, que se debe utilizar unha soa vez e logo débese pechar para unha soa unidade de traballo. Unha Session non obterá unha Connection JDBC a menos que sexa necesario. Non consumirá recursos ata que se utilice.

Unha transacción da base de datos ten que ser tan curta como sexa posible, para reducir a contención de bloqueos na base de datos. Así pois non se recomenda que se manteña unha transacción da base de datos aberta durante o tempo para pensar do usuario (cando teña que introducir datos), ata que a unidade de traballo se atope completa.