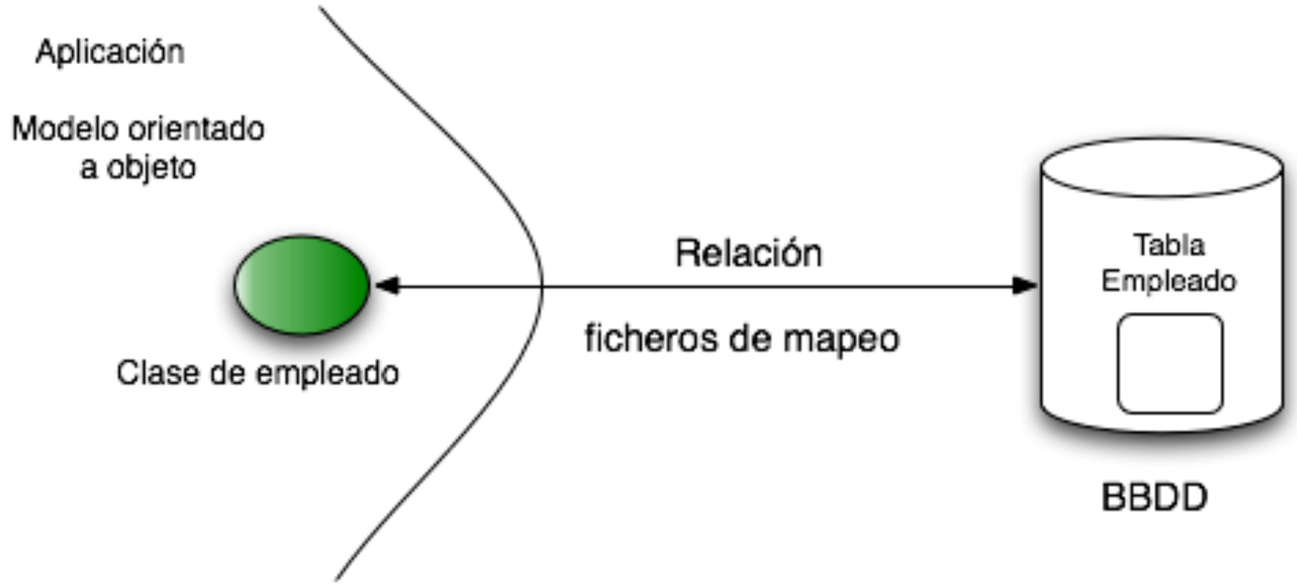
Trabajando con JPA

Entidades JPA

Índice del capítulo

Introducción

- Las **entidades** no son más que **POJOs que representan datos** que se pueden conservar en la base de datos.
- Una **entidad** representa una **tabla almacenada en una base de datos**.
- Cada **instancia de una entidad** representa una **fila en la tabla**.



Introducción

- Aprenderemos sobre los **conceptos básicos** de las entidades, junto con **varias anotaciones** que definen y personalizan una entidad.
- **JPA** necesita que estas clases cumplan **una serie de reglas para el funcionamiento óptimo** del sistema.
- Una instancia de una clase puede pasar por diferentes estados:
  - new, manage, detached y removed.
- El **conocimiento profundo del ciclo de vida** de la instancia de una clase persistente en JPA es **pieza clave**.

Reglas para la creación de clases persistentes

- Constructor sin parámetros.
- Propiedad identificadora.
- Clases no finales.
- Declaración de métodos consultores y modificadores.

¿Tiene que ser serializable?

- ¿Qué es la interfaz Serializable?
- Serializable y JPA.
- Exponiendo entidades en la capa de presentación.

¿Qué es la interfaz Serializable?

- Serializable es una de las pocas **interfaces** de tipo **marker** que se encuentran en el núcleo de Java. Las **interfaces de tipo markers** son interfaces de casos especiales **sin métodos ni constantes**.
- La **serialización de objetos** es el proceso de **convertir objetos Java en flujos de bytes**. Luego podemos transferir estos flujos de bytes por cable o almacenarlos en la memoria persistente.
- La **deserialización es el proceso inverso**, donde tomamos **flujos de bytes** y los convertimos nuevamente en objetos Java.
- Para permitir la **serialización** (o **deserialización**) de objetos, una **clase debe implementar la interfaz Serializable**. De lo contrario, nos encontraremos con java.io.NotSerializableException.
- La serialización se usa ampliamente en tecnologías como RMI, JPA y EJB.

Serializable y JPA

- Veamos qué dice la especificación JPA sobre Serializable y qué implicaciones tiene.
- Una de las partes centrales de JPA es una clase de entidad. Marcamos dichas clases como entidades (ya sea con la anotación @Entity o un descriptor XML). Hay varios requisitos que debe cumplir nuestra clase de entidad, y el que más nos preocupa, según la especificación JPA, es:
  - Si una **instancia de entidad** se va a pasar **por valor como un objeto separado** (por ejemplo, a través de una interfaz remota), la **clase de entidad debe implementar la interfaz Serializable**.
- En la práctica, si **nuestro objeto va a abandonar** el dominio de la JVM, requerirá **serialización**.
- Cada clase de entidad consta de **campos y propiedades persistentes**. La especificación requiere que los **campos de una entidad puedan ser primitivos de Java, tipos serializables de Java** o tipos **serializables** definidos por el usuario.

Serializable y JPA

- Una **clase de entidad** también debe tener una **clave principal**. Las claves primarias pueden ser **primitivas** (campo único persistente) o **compuestas**. Se aplican varias reglas a una clave compuesta, una de las cuales es que se requiere que una clave compuesta sea **serializable**.

```
@Entity
public class User {
   @EmbeddedId UserId userId;
   String email;
   // constructors, getters and setters
}

@Embeddable
public class UserId implements Serializable{
   private String name;
   private String lastName;
   // getters and setters
}
```

Exponiendo entidades en la capa de presentación

- Cuando enviamos objetos utilizando HTTP, generalmente creamos **DTO** (objetos de transferencia de datos) específicos para este propósito.

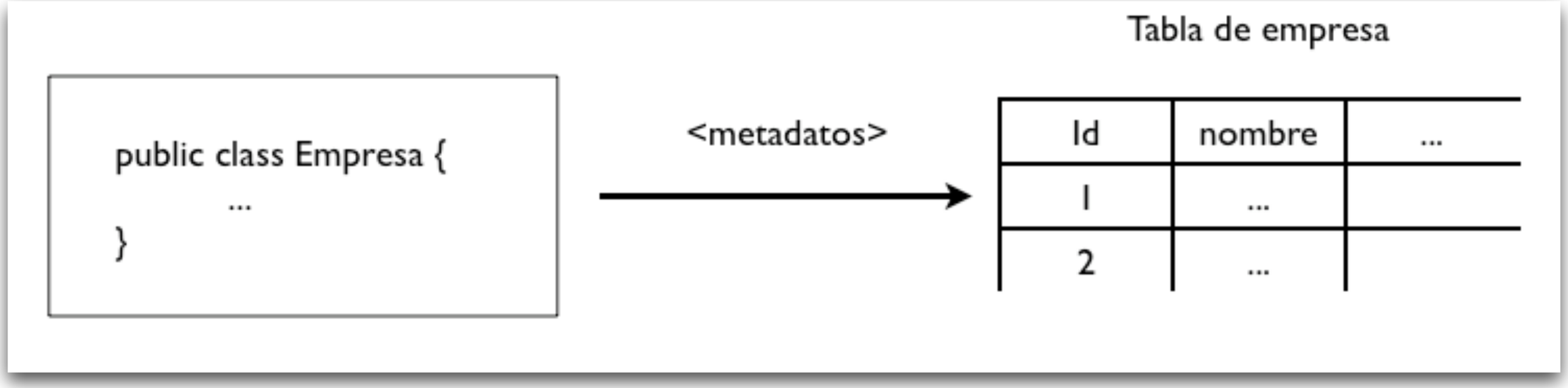
- Al crear DTO, **desacoplamos los objetos del dominio** interno de los servicios externos. Si queremos exponer nuestras entidades directamente a la capa de presentación sin DTO, entonces las **entidades deben ser serializables**.

- Utilizamos el **objeto HttpSession** para almacenar datos relevantes que nos ayudan a identificar a los usuarios en las visitas a varias páginas de nuestro sitio web.

- El **servidor web** puede almacenar datos de la sesión en disco cuando se apaga correctamente o transferir datos de la sesión a otro servidor web en entornos agrupados. Si una **entidad es parte de este proceso**, entonces debe ser **serializable**. De lo contrario, nos encontraremos con **NotSerializableException**.

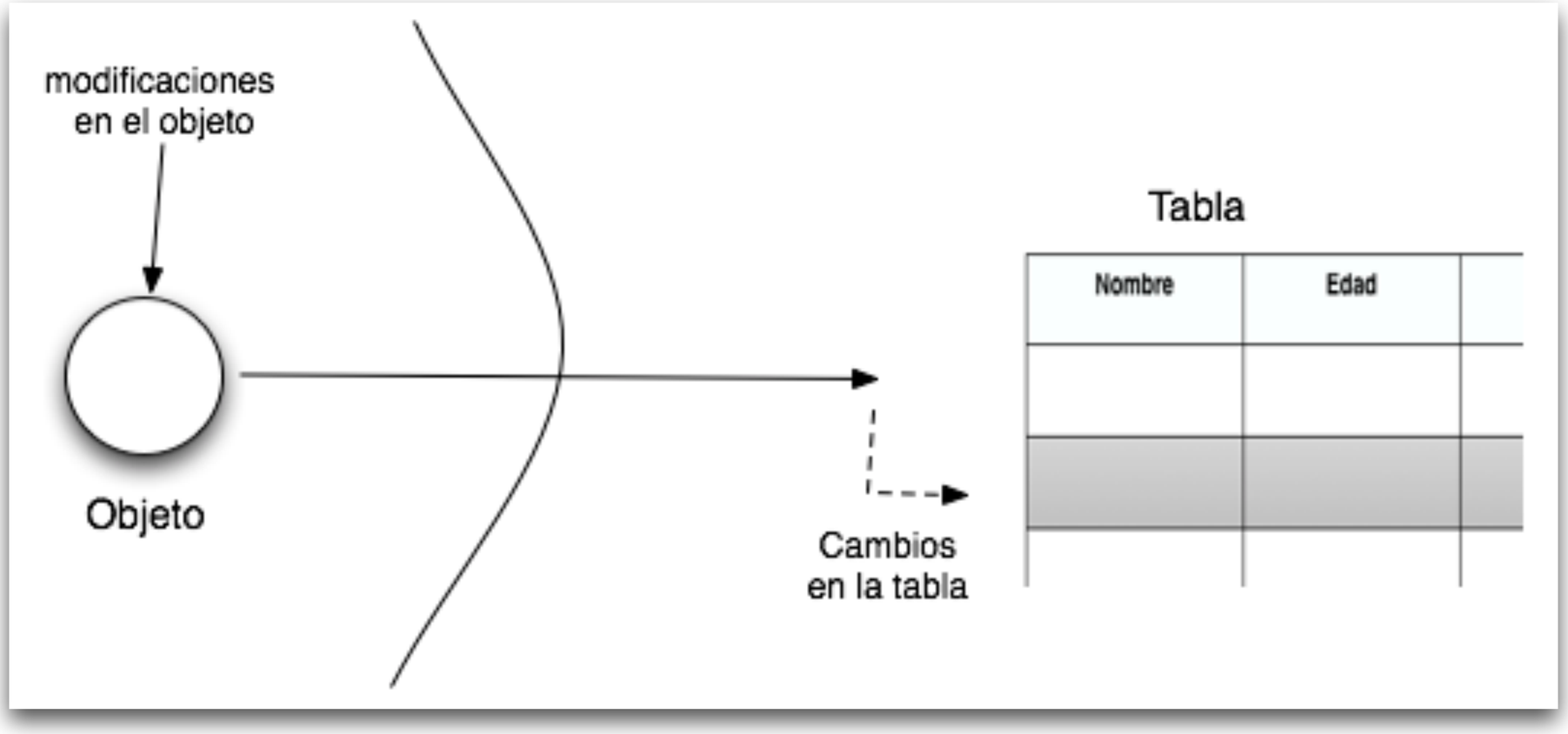
El mapeo de entidades

- Una vez que hemos definido una clase persistente, se pasa a describir uno de los conceptos más importantes de JPA, el **mapeo de clases persistentes** en una base de datos.



El mapeo de entidades

- Cuando mediante **JPA** se actúa sobre un **objeto asociado a una de estas clases persistentes**, provocará una serie de acciones sobre las tablas de la base de datos.



El mapeo de entidades

- Anotación @Entity.
- Anotación @Id.
- Anotación @Table.
- Anotación @Column.

Anotación @Entity

- Dado un POJO llamado Student, que representa los datos de un estudiante, y nos gustaría almacenarlo en la base de datos:

```
public class Student {
    // fields, getters and setters
}
```

- Para hacer esto, debemos definir una entidad para que JPA sea consciente de ella.

- Así que definámoslo haciendo uso de la **anotación @Entity**. Debemos especificar esta anotación a nivel de clase. También debemos asegurarnos de que la entidad tenga un constructor sin argumentos y una clave principal:

```
@Entity
public class Student {
    // fields, getters and setters
}
```

Anotación @Entity

- El **nombre de la entidad por defecto** es el **nombre de la clase**.
- Podemos cambiar su nombre usando el **atributo name**:

```
@Entity(name="student")
public class Student {

   // fields, getters and setters

}
```

- Debido a que varias implementaciones de JPA intentarán subclasificar nuestra entidad para proporcionar su funcionalidad, las clases de entidad no deben declararse finales.

Anotación @Id

- Cada **entidad JPA** debe tener una **clave principal que la identifique de forma única**.
- La **anotación @Id** define la **clave principal**.
- Podemos generar los identificadores de diferentes maneras, que se especifican mediante la **anotación @GeneratedValue**.
- Podemos elegir entre **cuatro estrategias de generación** de id con el elemento de estrategia. El valor puede ser AUTO, TABLE, SEQUENCE, or IDENTITY:

```
@Entity
public class Student {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;
   private String name;
   // getters and setters
}
```

Anotación @Table

- En la mayoría de los casos, el nombre de la tabla en la base de datos y el nombre de la entidad no serán iguales.
- En estos casos, podemos **especificar el nombre de la tabla** usando la **anotación @Table**:

```
@Entity
@Table(name="STUDENT")
public class Student {

   // fields, getters and setters

}
```

Anotación @Table

- También podemos mencionar el esquema usando el elemento schema:

```
@Entity
@Table(name="STUDENT", schema="SCHOOL")
public class Student {

   // fields, getters and setters

}
```

- El nombre del esquema ayuda a distinguir un conjunto de tablas de otro.
- Si **no usamos** la anotación @Table, el **nombre de la tabla** será el **nombre de la entidad**.

Anotación @Column

- Al igual que la **anotación @Table**, podemos usar la **anotación @Column** para definir los detalles de una columna en la tabla. La anotación @Column tiene muchos atributos, como **name, length, nullable, and unique**:

```
@Entity
@Table(name="STUDENT")
public class Student {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;

   @Column(name="STUDENT_NAME", length=50, nullable=false, unique=false)
   private String name;

   // other fields, getters and setters
}
```

Anotación @Column

- El atributo **name** especifica el **nombre de la columna en la tabla**.
- El atributo **length** especifica su longitud.
- El atributo **nullable** especifica si la columna es null o no, y el **atributo unique** especifica si la columna es única.
- Si **no especificamos** esta anotación, el nombre de la columna en la tabla será el **nombre del campo**.

Gestión de la persistencia

- Introducción.
- Estado de las entidades.
- Operaciones del EntityManager.

Introducción

- Entender el **ciclo de vida de las entidades** es sin duda uno de los puntos cruciales de JPA, pues entender cómo gestiona EntityManager nos permitirá entender mejor **cómo funciona JPA** y prevenir muchos errores en tiempo de ejecución.

- Lo primero que debemos entender, es que todas entidades que utilicemos con JPA, serán **administradas por el EntityManager**.

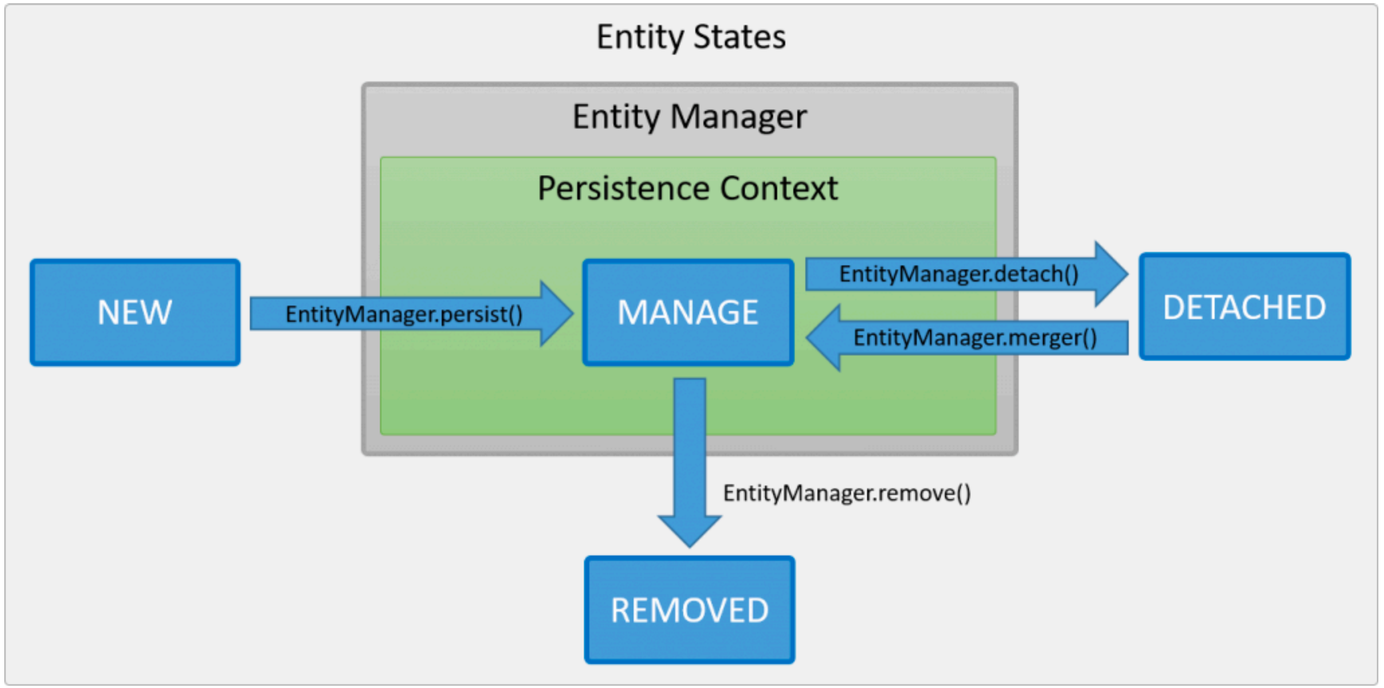
Estado de las entidades

- Introducción.
- Estado new.
- Estado manage.
- Estado detached.
- Estado removed.

Introducción

- A medida que trabajamos con las entidades, **estas van cambiando de estado** y el estado de estas será utilizado para **realizar operaciones** en la base de datos.

- Un aspecto importante cuando utilizamos JPA es que ya **no estamos trabajando con sentencias SQL** y en su lugar trabajamos con las **Entidades y sus estados**. Las entidades en JPA puedan pasar por **cuatro estados distintos** que se pueden ver en la siguiente imagen:



Introducción

- Los cuatro estados son:
  - **New**: estado que tiene una entidad cuando es creada **con el operador new,** por lo tanto, no existe en la base de datos y no está asociada a un contexto de persistencia. Todas las entidades en estado new no serán afectadas en la base de datos cuando la transacción finalice, pues no está asociado a al Context Persistence.
  - **Manage**: Todas las entidades que están siendo administradas por JPA están en **estado manage**, lo que indica que son parte de un **contexto de persistencia**. Aplica para entidades que serán insertadas, actualizadas o fueron obtenidas por el EntityManager como parte de alguna operación SELECT.
  - **Detached**: Las entidades en estado detached son entidades que **sí existen en la base de datos**, pero por algún motivo **no son parte de un contexto de persistencia**. Todas las entidades en este estado no serán afectadas en la base de datos, pues no pertenecen a un Persistence Context
  - **Removed**: Cuando una entidad es marcada para ser **eliminada**, pasa automáticamente al estado de **Removed**. Cuando la transacción finaliza, todas las entidades en este estado serán **eliminadas** de la base de datos y **ya no existirán más** en el Persistence Context.

Estado new

- La información de **un objeto new** no se encuentran almacenados en la base de datos.
- Los objetos creados mediante el operador new no son inmediatamente persistentes, su estado es transitorio.
- Esto significa que no están asociados a ninguna base de datos.

```
// Objeto en estado transient
Empleado e = new Empleado(“Marta”,34);
```

Estado manage

- Es aquel que se encuentra asociado a una identidad en una base de datos.
- Un objeto persistente puede ser:
  - Un objeto creado por la aplicación y posteriormente hecho persistente llamando al método persist del EntityManager.
  - Un objeto que ha sido creado mediante una referencia creada a partir de otro objeto persistente.
  - Un objeto que ha sido devuelto desde la base de datos gracias a la **ejecución de una consulta**.
- Un **objeto en estado manage** está sincronizado con la base de datos.
- Cualquier **modificación** que se realice sobre ese objeto será tenida en cuenta **en la base de datos**.

Estado detached

- La asociación entre el gestor de persistencia y los objetos persistentes desaparece cuando se llama al **método detach() del EntityManager**.

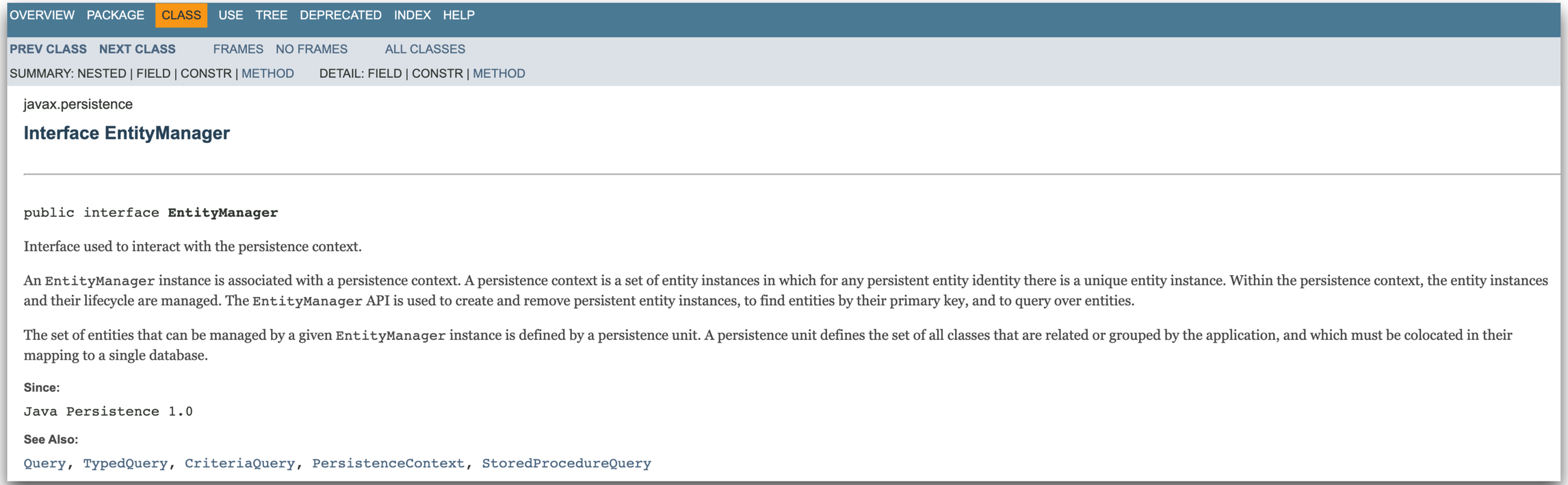
- A partir de ese momento, esos objetos pasan a llamarse independientes (detached), su estado ya no sincronizará con el estado de la base de datos.

Operaciones del EntityManager

- Introducción.
- Transacciones.
- Persist para hacer persistente una entidad.
- Find para buscar entidades.
- Merge.
- Remove para borrar entidades.
- Clear.
- Actualización de entidades.
- Flush.
- Queries.

Introducción

- La API de la interfaz **EntityManager** define todas las operaciones que debe implementar un entity manager:



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

javax.persistence

Interface EntityManager

`public interface EntityManager`

Interface used to interact with the persistence context.

An `EntityManager` instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The `EntityManager` API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

The set of entities that can be managed by a given `EntityManager` instance is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be colocated in their mapping to a single database.

Since:

`Java Persistence 1.0`

See Also:

`Query`, `TypedQuery`, `CriteriaQuery`, `PersistenceContext`, `StoredProcedureQuery`

Introducción

- Destacamos las siguientes:
  - void **clear**(): borra el contexto de persistencia, desconectando todas sus entidades.
  - boolean **contains**(Object entity): comprueba si una entidad está gestionada en el contexto de persistencia.
  - Query **createNamedQuery**(String name): obtiene una **consulta JPQL precompilada**.
  - void **detach**(Object entity): elimina la entidad del contexto de persistencia, dejándola desconectada de la base de datos.
  - <T> T **find**(Class<T>, Object key): busca por **clave primaria**.

Introducción

- (cont.):
  - void **flush**(): sincroniza el contexto de persistencia con la base de datos.
  - <T> T **getReference**(Class<T>, Object key): obtiene una referencia a una entidad, que puede haber sido recuperada de forma **lazy**.
  - EntityTransaction **getTransaction**(): devuelve la **transacción** actual.
  - <T> T **merge**(T entity): incorpora una entidad al contexto de persistencia, haciéndola gestionada.
  - void **persist**(Object entity): hace una entidad persistente y gestionada.
  - void **refresh**(Object entity): refresca el estado de la entidad con los valores de la base de datos, sobrescribiendo los cambios que se hayan podido realizar en ella.
  - void **remove**(Object entity): elimina la entidad.

Transacciones

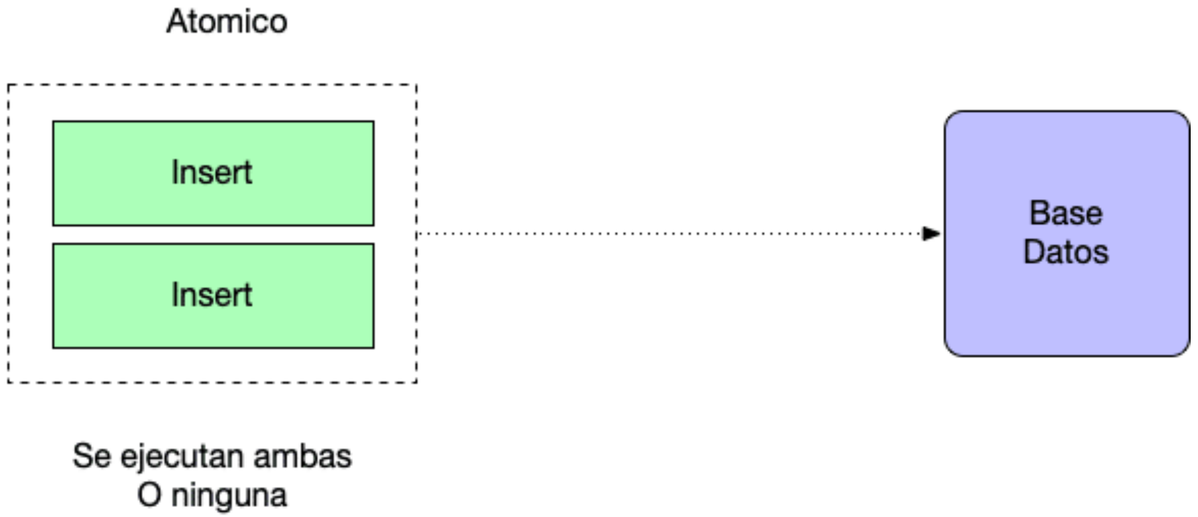
- Introducción.
- Transacciones en JPA.
- Ejemplo.

Introducción

- El **concepto de JPA Transaction** es uno de los conceptos claves cuando uno comienza a trabajar con JPA (Java Persistence API).
- Sin embargo una transacción es un concepto que pertenece más a las bases de datos que a JPA propiamente dicho.
- Una transacción hace referencia a una operación que se realiza contra una base de datos y que cumple con las **propiedades ACID**.
- Pasamos a describirlas a continuación.

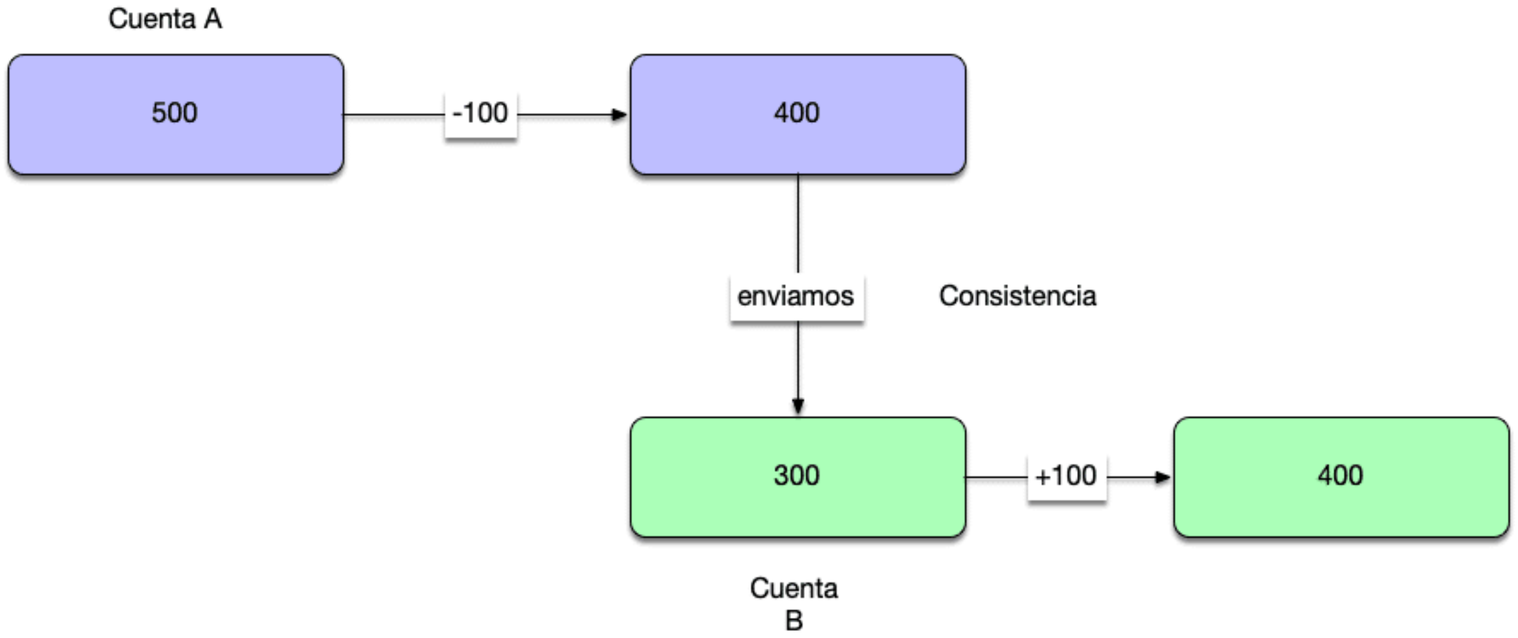
Introducción

- **Atomic**:
  - Habla de cuando una operación es **indivisible** o se realiza de forma completa o no se realiza ninguna de sus partes.
  - Esto es muy habitual en base de datos ya que si, por ejemplo, queremos realizar dos inserciones si las ejecutamos dentro de una transacción o se realizarán ambas o no se realizará ninguna. Cualquier **fallo** hará que la transacción se **revierta** y no se aplique ninguna modificación.



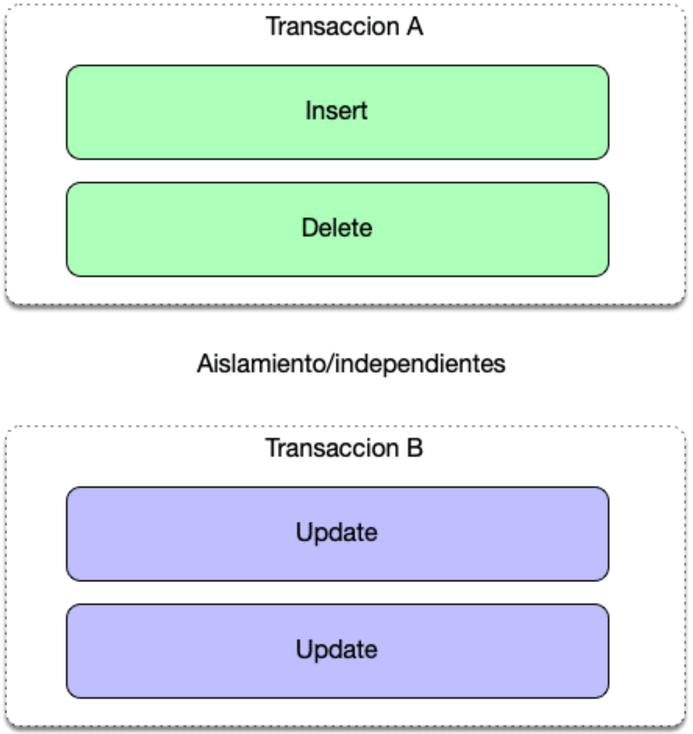
Introducción

- **Consistente**:
  - Si, por ejemplo, realizamos **dos modificaciones en la base de datos** una que me elimina 100 euros de mi saldo y otra que inserta 100 euros en otra cuenta . El concepto de consistencia hace referencia a que los números deben cuadrar si resto 100 euros de mi cuenta en la otra deben de entrar 100 no pueden entrar 70.



Introducción

- **Isolated/Aislada**:
  - La operación que realizamos **no puede verse afectada** por otras operaciones que se estén ejecutando en ese momento:



Introducción

- **Durable**:
  - La información queda grabada en la base de datos para el futuro y persiste.

Transacciones en JPA

- Cualquier operación que conlleve una **creación, modificación o borrado de entidades** debe hacerse dentro de una **transacción**.
- En **JPA las transacciones** se gestionan de forma distinta dependiendo de si estamos en un entorno **Java SE o en un entorno Java EE**.
- La diferencia fundamental entre ambos casos es que en un entorno Java EE las transacciones se manejan con **JTA (Java Transaction API)**, una API que implementa **el two face commit** y que permite gestionar operaciones **sobre múltiples recursos transaccionales** o múltiples operaciones transaccionales sobre el mismo recurso.
- En el caso de **Java SE** las transacciones se implementan con el gestor de **transacciones propio del recurso local** (la base de datos) y se especifican en la interfaz **EntityTransaction**.

Transacciones en JPA

- El **gestor de transacciones locales** se obtiene con la llamada **getTransaction()** al EntityManager.

getTransaction

```
EntityTransaction getTransaction()
```

Return the resource-level `EntityTransaction` object. The `EntityTransaction` instance may be used serially to begin and commit multiple transactions.

Returns:

`EntityTransaction instance`

Throws:

`IllegalStateException` – if invoked on a JTA entity manager

- El tipo EntityTransaction nos ofrece toda la funcionalidad que vamos a necesitar.

Transacciones en JPA

- Una vez obtenido, podemos pedirle cualquiera de los métodos definidos en la interfaz:
  - **begin()** para comenzar la transacción,
  - **commit()** para actualizar los cambios en la base de datos (en ese momento JPA vuelca las sentencias SQL en la base de datos) o
  - **rollback()** para deshacer la transacción actual.

Ejemplo

```
EntityManager em= ...
Libro libro2= new Libro ("6B","Spring","Gema");
Libro libro3= new Libro ("4A","Java","David");
EntityTransaction tx = em.getTransaction();
try {
   tx.begin();
   em.persist(libro2);
   em.persist(libro3);
   tx.commit();
} catch (Exception e) {
   tx.rollback();
}
```

Persist para hacer persistente una entidad

- El **método persist() del EntityManager** acepta una **nueva instancia de entidad** y la convierte en gestionada/managed.
- Si la entidad que se pasa como parámetro ya está gestionada en el contexto de persistencia, la llamada se ignora. La **operación contains()** puede usarse para comprobar si una entidad está gestionada.
- El hecho de convertir una **entidad en gestionada** no la hace **persistir** inmediatamente en la **base de datos**. La **verdadera llamada a SQL** para crear los datos relacionales **no se generará hasta que el contexto de persistencia se sincronice** con la base de datos. Lo más normal es que esto suceda cuando se realiza un **commit de la transacción**.
- En el momento en que la entidad se convierte en **gestionada**, los cambios que se realizan sobre ella afectan al **contexto de persistencia**. Y en el momento en que la **transacción termina**, el estado en el que se encuentra la entidad **es volcado en la base de datos**.

Persist para hacer persistente una entidad

- Si se llama a **persist() fuera** de una **transacción** la entidad se incluirá en el contexto de persistencia, pero **no se realizará ninguna acción** hasta que la **transacción comience** y el contexto de persistencia se sincronice con la base de datos.

- La **operación persist()** se utiliza con **entidades nuevas** que **no existen en la base de datos**.

- Si se le pasa una instancia con un **identificador que ya existe** en la base de datos el proveedor de persistencia puede detectarlo y **lanzar una excepción EntityExistsException**. Si no lo hace, entonces se lanzará la **excepción cuando se sincronice el contexto de persistencia** con la base de datos, al encontrar una **clave primaria duplicada**.

Persist para hacer persistente una entidad

- Ejemplo:

```
Departamento dept = em.find(Departamento.class, 30);
Empleado emp = new Empleado();
emp.setNombre("Pedro");
emp.setDepartamento(dept);
dept.getEmpleados().add(emp);
em.persist(emp);
```

- En el ejemplo comenzamos **obteniendo una instancia que ya existe** en la base de datos de la entidad **Departamento**.
- Se crea una **nueva instancia de Empleado**, proporcionando algún atributo.
- Después asignamos el empleado al departamento, llamando al **método setDepartamento()** del empleado y pasándole la instancia de Departamento que habíamos recuperado.

Persist para hacer persistente una entidad

- Actualizamos el otro lado de la relación **llamando al método add() de la colección** para que el contexto de persistencia mantenga correctamente la relación bidireccional.
- Y por último, realizamos **la llamada al método persist()** que convierte la entidad en gestionada.
- Cuando el contexto de persistencia se sincroniza con la base de datos, se añade la nueva entidad en la tabla y se actualiza al mismo tiempo la relación.
- Hay que hacer notar que **solo se actualiza la tabla de Empleado**, que es la propietaria de la relación y la que contiene la clave ajena a **Departamento**.

Persist para hacer persistente una entidad

- El código asume la **existencia de un EntityManager** en la variable de instancia **em** y lo usa para hacer persistente el empleado recién creado.
- La entidad se vuelca realmente a la base de datos **cuando se realiza un flush del contexto de persistencia** y se generan las instrucciones SQL que se ejecutan en la base de datos (normalmente al hacer un commit de la transacción actual).
- Si el EntityManager encuentra **algún problema al ejecutar el método**, se lanza la excepción no chequeada **PersistenceException**. Cuando termine la ejecución del método, si no se ha producido ninguna excepción, emp será a partir de ese momento una entidad gestionada dentro del contexto de persistencia del EntityManager.
- Una cuestión a tener en cuenta es que el **identificador del empleado** puede no estar disponible en un tiempo, hasta que se realice el flush del contexto de persistencia.

Persist para hacer persistente una entidad

- Una mejora del código anterior, que garantiza que el identificador del empleado se carga en la entidad sería la siguiente:

```
Departamento dept = em.find(Departamento.class, 30);
Empleado empleado = new Empleado();
empleado.setNombre("Pedro");
empleado.setDepartamento(dept);
dept.getEmpleados().add(empleado);
em.persist(empleado);
em.flush(empleado);
em.refresh(empleado);
```

- La llamada a **flush** asegura que se ejecuta el insert en la BD y la llamada a refresh asegura que el identificador se carga en la instancia.

Find para buscar entidades

- Una vez que la entidad está en la base de datos, lo siguiente que podemos hacer es **recuperarla de nuevo**. Para ello basta con escribir una línea de código:

```
Empeado empleado = em.find(Empleado.class, 146);
```

- Pasamos la **clase de la entidad** que estamos buscando (en el ejemplo estamos buscando una instancia de la clase Empleado) y **el identificador o clave primaria** que identifica la entidad.

- El entity manager buscará esa entidad en la **base de datos y devolverá** la instancia buscada. La entidad devuelta será una **entidad gestionada** que existirá en el contexto de persistencia actual asociado al entity manager.

- En el caso en que **no existiera ninguna entidad** con ese identificador, se devolvería **simplemente null**.

Find para buscar entidades

- La llamada a **find** puede devolver **dos posibles excepciones de tiempo de ejecución**, ambas de la clase PersistenceException: **IllegalStateException** si el entitiy manager ha sido previamente cerrado o **IllegalArgumentException** si el primer argumento no contiene una clase entidad o el segundo no es el tipo correcto de la clave primaria de la entidad.

- Existe una **versión especial de find()** que sólo recupera **una referencia a la entidad**, sin obtener los datos de los campos de la base de datos. Se trata del **método getReference()**.

- Es útil cuando se quiere añadir **un objeto con una clave primaria conocida a una relación**. Ya que únicamente estamos creando una relación, **no es necesario cargar todo el objeto** de la base de datos.

- Sólo se necesita su **clave primaria**.

Find para buscar entidades

- Veamos la nueva versión del ejemplo anterior:

```
Departamento dept = em.getReference(Departamento.class, 30);
Empleado emp = new Empleado();
emp.setId(53);
emp.setNombre("Pedro");
emp.setDepartamento(dept);
dept.getEmpleados().add(emp);
em.persist(emp);
```

- Esta versión es **más eficiente que la anterior** porque **no se realiza ningún SELECT** en la base de datos para buscar **la instancia del Departamento**.

- Cuando se llama a getReference(), el proveedor devolverá un **proxy al Departamento** sin recuperarlo realmente de la base de datos. En tanto que sólo se acceda a la clave primaria, **no se recuperará ningún dato**. Y cuando se haga persistente el Empleado, se guardará en la clave ajena correspondiente el valor de la clave primaria del Departamento.

Find para buscar entidades

- Un posible problema de este método es que, a diferencia de find() **no devuelve null** si la instancia no existe, ya que realmente no realiza la búsqueda en la base de datos.

- Únicamente se debe utilizar el método **cuando estamos seguros de que la instancia existe** en la base de datos. En caso contrario estaremos guardando en la **variable dept una referencia** (clave primaria) de una entidad **que no existe**, y cuando se haga persistente el empleado **se generará una excepción** porque el Empleado estará haciendo referencia a una entidad no existente.

- En general, la **mayoría de las veces llamaremos al método find()** directamente. Las implementaciones de JPA hacen un buen trabajo con las cachés y si ya tenemos la entidad en el contexto de persistencia no se realiza la consulta a la base de datos.

Merge

- El **método merge()** permite volver a **incorporar en el contexto de persistencia** del entity manager una entidad que había sido **desconectada**.
- Debemos pasar como **parámetro la entidad que queremos incluir**.
- Hay que tener **cuidado con su utilización**, porque el objeto que se pasa como parámetro no pasa a ser gestionado. Hay que usar **el objeto que devuelve el método**.
- Ejemplo:

```
public void subeSueldo(Empleado emp, long inc){
  ...
  Empleado empGestionado = em.merge(emp);
  empGestionado.setSueldo(empGestionado.getSueldo()+inc);
}
```

Merge

- Si una entidad con **el mismo identificador** que emp existe en el **contexto de persistencia**, se devuelve como resultado y se actualizan sus atributos.

- Si el objeto que se le pasa a merge() es **un objeto nuevo**, se comporta **igual que persist()**, con la única diferencia de que la entidad gestionada es la **devuelta como resultado de la llamada**.

Remove para borrar entidades

- Un **borrado** de una entidad realiza una **sentencia DELETE en la base de datos**.
- Esta acción **no es demasiado frecuente**, ya que las aplicaciones de gestión normalmente conservan todos los datos obtenidos y marcan como no activos aquellos que quieren dejar fuera de vista de los casos de uso. Se suele utilizar para **eliminar datos que se han introducido por error en la base de datos** o para trasladar de una tabla a otra los datos (se borra el dato de una y se inserta en la otra).
- En el caso de entidades esto último sería equivalente a un cambio de tipo de una entidad.
- Para **eliminar** una entidad, la **entidad debe estar gestionada**, esto es, debe existir en el contexto de persistencia.
- Esto significa que la aplicación **debe obtener la entidad antes de eliminarla**.

Remove para borrar entidades

- Un ejemplo sencillo es:

```
Empleado empleado = em.find(Empleado.class, 146);
em.remove(emp);
```

- La llamada a **remove** asume que el **empleado existe**. En el caso de no existir **se lanzaría una excepción**.
- Borrar una entidad **no es una tarea compleja**, pero puede requerir algunos pasos, dependiendo del número de relaciones en la entidad que vamos a borrar.
- En su forma más simple, el **borrado de una entidad** se realiza pasando la **entidad como parámetro** del método remove() del entity manager que la gestiona.
- En el momento en que el **contexto de persistencia se sincroniza** con una transacción y se realiza un commit, **la entidad se borra**. Hay que tener cuidado, sin embargo, con las relaciones en las que participa la entidad para no comprometer la integridad de la base de datos.

Remove para borrar entidades

- Veamos un sencillo ejemplo.
- Consideremos las **entidades Empleado y Despacho** y supongamos una relación unidireccional **uno-a-uno entre Empleado y Despacho** que se mapea utilizando una clave ajena en la tabla **EMPLEADO** hacia la tabla **DESPACHO**.
- Supongamos el siguiente código dentro de una transacción en el que borramos el despacho de un empleado:

```
Empleado emp = em.find(Empleado.class, empId);
Despacho desp = emp.getDespacho();
em.remove(desp);
```

Remove para borrar entidades

- Cuando se realice un commit de la transacción veremos una **sentencia DELETE** en la tabla DESPACHO, pero en ese momento obtendremos una **excepción con un error de la base de datos** referido a que hemos **violado una restricción de la clave ajena**.

- Esto se debe a que existe una **restricción de integridad referencial** entre la tabla EMPLEADO y la tabla DESPACHO.

- Se ha borrado una fila de la tabla **DESPACHO** pero la clave ajena correspondiente en la tabla EMPLEADO **no se ha puesto a NULL**. Para corregir el problema, debemos poner explícitamente a null el atributo despacho de la entidad Empleado antes de que la transacción finalice:

```
Empleado emp = em.find(Empleado.class, empId);
Despacho desp = emp.getDespacho();
emp.setDespacho(null);
em.remove(desp);
```

Remove para borrar entidades

- El mantenimiento de las relaciones es **responsabilidad de la aplicación**.
- Casi todos los problemas que suceden en los borrados de entidades tienen relación con este aspecto.
- Si la entidad que se va a **borrar es el objetivo de una clave ajena** en otras tablas, entonces debemos **limpiar esas claves ajenas** antes de borrar la entidad.

Clear

- En ocasiones puede **ser necesario limpiar (clear) contexto de persistencia** y vaciar las entidades gestionadas.
- Esto puede suceder, por ejemplo, en contextos extendidos gestionados por la aplicación que han crecido demasiado.
- Por ejemplo, consideremos el caso de **un entity manager** gestionado por la aplicación que lanza una consulta que **devuelve varios cientos de instancias entidad**.
- Una vez que ya hemos realizado los cambios a unas cuantas de esas instancias y la transacción se termina, **se quedan en memoria cientos de objetos** que no tenemos intención de cambiar más.
- Si **no queremos cerrar el contexto** de persistencia en ese momento, entonces **tendremos que limpiar** de alguna forma las instancias gestionadas, o el contexto de persistencia irá creciendo cada vez más.

Clear

- El **método clear() del interfaz EntityManager** se utiliza para limpiar el contexto de persistencia.
- En muchos sentidos su funcionamiento es similar a un rollback de una transacción en memoria.
- **Todas las instancias gestionadas** por el contexto e persistencia **se desconectan** del contexto y quedan con el estado previo a la llamada a clear().
- La operación clear() es del **tipo todo o nada**.
- **No es posible cancelar selectivamente** la gestión de una instancia particular cuando el contexto de persistencia está abierto.

Actualización de entidades

- Para actualizar una entidad, primero debemos obtenerla **para convertirla en gestionada**. Después podremos **colocar los nuevos valores en sus atributos** utilizando los métodos set de la entidad.

- Por ejemplo, supongamos que queremos subir el sueldo del empleado 146 en 1.000 euros. Tendríamos que hacer lo siguiente:

```
...
Empleado empleado = em.find(Empleado.class, 146);
empleado.setSueldo(empleado.getSueldo() + 1000);
```

- Nótese la diferencia con las operaciones anteriores, en las que el EntityManager era el responsable de realizar la operación directamente. Aquí **no llamamos al EntityManager** sino a la propia entidad. Estamos, por así decirlo, trabajando con una caché de los datos de la base de datos.

- Posteriormente, cuando **se finalice la transacción**, el EntityManager hará persistentes los cambios mediante las correspondientes sentencias SQL.

Actualización de entidades

- La **otra forma de actualizar** una entidad es con el **método merge() del EntityManager**.
- A este método se le pasa **como parámetro una entidad no gestionada**.
- El EntityManager busca la entidad en su contexto de persistencia (utilizando su **identificador**) y actualiza **los valores del contexto de persistencia con los de la entidad no gestionada**. En el caso en que la entidad no existiera en el contexto de persistencia, se crea con los valores que lleva la entidad no gestionada.
- Es **muy importante notar** que **no está permitido modificar la clave primaria** de una entidad gestionada. Si intentamos hacerlo, en el momento de hacer un commit la transacción lanzará una **excepción RollbackException**. Para reforzar esta idea, es conveniente **definir las entidades sin un método set de la clave primaria**. En el caso de aquellas entidades con una generación automática de la clave primaria, ésta se generará en tiempo de creación de la entidad. Y en el caso en que la aplicación tenga que proporcionar la clave primaria, lo puede hacer en el constructor.

Flush

- El **método flush()** se utiliza para **sincronizar** el **contexto** de persistencia con la **base de datos**. El contexto de persistencia en JPA es responsable de realizar operaciones de lectura y escritura en la base de datos en nombre de las entidades administradas. Sin embargo, estas **operaciones no se traducen inmediatamente** en consultas SQL ejecutadas en la base de datos.

- Cuando invocas el método flush(), JPA realiza las siguientes acciones:

  - **Sincronización** con la base de datos: Los cambios realizados en las entidades administradas se sincronizan con la base de datos. Esto implica que las instrucciones SQL necesarias para reflejar esos cambios se envían a la base de datos.

  - **Ejecución de SQL**: Los comandos SQL generados para las operaciones pendientes (como inserciones, actualizaciones o eliminaciones) se ejecutan en la base de datos.

  - **Actualización de la cache**: El contexto de persistencia actualiza su caché con los resultados de las operaciones recientes en la base de datos.

Flush

- Es importante destacar que **el método flush()** no confirma la transacción, ya que eso se logra mediante el **método commit()** en el caso de transacciones manuales.
- Si estás utilizando **transacciones gestionadas por el contenedor** (por ejemplo, en un entorno Java EE), la transacción se manejará automáticamente.
- El uso de flush() **puede ser útil** en situaciones donde necesitas asegurarte de que los cambios realizados en el contexto de persistencia se reflejen inmediatamente en la base de datos, en lugar de esperar hasta el final de la transacción.
- Sin embargo, en la **mayoría de los casos,** el flush() no es necesario, ya que **JPA maneja automáticamente la sincronización** con la base de datos al finalizar la transacción.

Queries

- Uno de los aspectos fundamentales de JPA es la posibilidad de **realizar consultas** sobre las entidades, muy similares a las consultas SQL. El lenguaje en el que se realizan las consultas se denomina **Java Persistence Query Language** (JPQL).

- Una consulta se implementa mediante un **objeto Query**. Los objetos Query se construyen utilizando el EntityManager como una factoría. La interfaz EntityManager proporciona un conjunto de métodos que devuelven un objeto Query nuevo.

- Una consulta puede ser **estática o dinámica**.

- Las **consultas estáticas** se definen con metadatos en forma de anotaciones o XML, y deben incluir la consulta propiamente dicha y un nombre asignado por el usuario. Este tipo de consulta se denomina una **consulta con nombre** (named query en inglés). El **nombre** se utiliza en tiempo de ejecución para **recuperar la consulta**.

Queries

- Una **consulta dinámica** puede lanzarse en **tiempo de ejecución** y **no es necesario darle un nombre**, sino especificar únicamente las condiciones.

- Son un poco **más costosas de ejecutar**, porque el proveedor de persistencia (el gestor de base de datos) no puede realizar ninguna preparación, pero son **muy útiles y versátiles** porque pueden construirse en función de la lógica del programa, o incluso de los datos proporcionados por el usuario.

Queries

- Ejemplo:

```
Query query = em.createQuery("SELECT e FROM Empleado e " +
                             "WHERE e.sueldo > :sueldo");
query.setParameter("sueldo", 20000);
List emps = query.getResultList();
```

- En el ejemplo vemos que, al igual que en JDBC, es posible **especificar consultas con parámetros** y posteriormente especificar esos parámetros con el método setParameter().

- Una vez definida la consulta, el **método getResultList()** devuelve la lista de entidades que cumplen la condición. Este método devuelve un objeto que implementa la interfaz List, una subinterfaz de Collection que soporta ordenación.

- Hay que notar que **no se devuelve una List<Empleado>** ya que no se pasa ninguna clase en la llamada y no es posible parametrizar el tipo devuelto.

Queries

- Sí que podemos hacer un casting en los valores devueltos por los métodos que implementan las búsquedas, como muestra el siguiente código:

```
public List<Empleado> findEmpleadosSueldo(long sueldo) {
  Query query = em.createQuery("SELECT e FROM Empleado e " +
                "WHERE e.sueldo > :sueldo");
  query.setParameter("sueldo", 20000);
  return (List<Empleado>) query.getResultList();
}
```

Otras anotaciones importantes

- Más anotaciones: @Transient, @Temporal y @Enumerated.
- Valores por defecto en las entidades.
- Entidades embebidas: @Embedded y @Embeddable.
- Trabajando con valores nulos.
- Trabajando con tipos básicos.
- Trabajando con objetos pesados.

Anotación @Transient

- A veces, es posible que queramos hacer que **un campo no sea persistente**. Podemos usar la **anotación @Transient** para hacerlo.
- Ejemplo:

```
public class Professor {
 @Id private int id;
 private String name;
 @Transient private String convertedName;
 ...
 public void setName(String name) {
    this.name = name;
    convertedName = convertName(name);
 }
 protected String convertName(String name) {
   return name.toUpperCase(Locale.CANADA);
 }
```

Anotación @Transient

- Así que anotamos el campo age con la anotación @Transient:

```
@Entity
@Table(name="STUDENT")
public class Student {
   ...
   @Column(name="STUDENT_NAME", length=50, nullable=false)
   private String name;

   @Transient
   private Integer age;
  ...
}
```

- Como resultado, el campo age no se guardará en la tabla.

Anotación @Temporal

- Esta anotación debe especificarse para **campos persistentes o propiedades** de tipo **java.util.Date** y **java.util.Calendar**. Sólo se puede especificar para campos o propiedades **de estos tipos**.
- La **anotación Temporal** se puede utilizar junto con la anotación **Basic**, la anotación **Id** o la anotación **ElementCollection** (cuando el valor de la colección de elementos es de dicho tipo temporal).
- En las API de Java simples, la precisión temporal del tiempo no está definida. Cuando se trata de datos temporales, es posible que desee describir la precisión esperada en la base de datos. Los datos temporales pueden tener precisión de DATE, TIME o TIMESTAMP (es decir, la fecha real, solo la hora o ambas).
- Utilice la **anotación @Temporal** para ajustar eso.

Anotación @Temporal

- Los datos temporales son los datos relacionados con el tiempo. Por ejemplo, en un sistema de gestión de contenidos, la fecha de creación y la fecha de última actualización de un artículo son datos temporales. En algunos casos, los datos temporales necesitan precisión y desea almacenar la fecha/hora precisas o ambas (TIMESTAMP) en la tabla de la base de datos.

- La precisión temporal no se especifica en las API principales de Java. @Temporal es una anotación JPA que convierte entre marca de tiempo y java.util.Date. También convierte la marca de tiempo en tiempo.

- Por ejemplo, en el siguiente fragmento, @Temporal(TemporalType.DATE) elimina el valor de hora y solo conserva la fecha.

Anotación @Temporal

- En algunos casos, es posible que tengamos que guardar valores temporales en nuestra tabla. Para esto, tenemos la **anotación @Temporal**:

```
@Entity
@Table(name="STUDENT")
public class Student {
  ...
   @Temporal(TemporalType.DATE)
   private Date birthDate;

}
```

- Sin embargo, con JPA 2.2, también tenemos soporte para java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime y java.time.OffsetDateTime.

Anotación @Enumerated

- A veces, es posible que deseemos persistir un tipo enumerado Java.
- En realidad, no tenemos que especificar la anotación @Enumerated en absoluto si vamos a persistir el Género por el ordinal del enumerado.

```
public enum Gender {
   MALE,
   FEMALE
}
```

```
public class Student {
   ...
  private Gender gender;
}
```

Anotación @Enumerated

- Podemos usar la anotación @Enumerated para especificar si la enumeración debe persistir por nombre o por ordinal (por defecto):

- Sin embargo, para conservar gender **por nombre del enumerado**, debemos incluir la anotación con **EnumType.STRING**.

```
@Entity
@Table(name="STUDENT")
public class Student {
   ...
   @Enumerated(EnumType.STRING)
   private Gender gender;
}
```

Valores por defecto en las entidades

- Vamos a ver los **valores por defecto de las columnas** de una tabla en JPA. Aprenderemos cómo configurar dichos valores como una **propiedad por defecto en la entidad**, así como directamente en la **definición de la tabla SQL**.

- La primera forma de establecer un valor de columna por defecto es establecerlo directamente como un valor de propiedad de entidad:

```
@Entity
public class User{
   @Id
   private Long id;
   private String firstName = "John Snow";
   private Integer age = 25;
   private Boolean locked = false;
}
```

Valores por defecto en las entidades

- Ahora, cada vez que creamos una entidad usando el **operador new**, establecerá los valores por defecto que proporcionamos:

```
@Test
void saveUser_shouldSaveWithDefaultFieldValues(){
    User user = new User();
    user = userRepository.save(user);

    assertEquals(user.getName(), "John Snow");
    assertEquals(user.getAge(), 25);
    assertFalse(user.getLocked());
}
```

Valores por defecto en las entidades

- Hay un **inconveniente** en esta solución. Cuando echamos un vistazo a la **definición de la tabla SQL**, no veremos ningún valor por defecto en ella:

```
create table user
(
    id     bigint not null constraint user_pkey primary key,
    name   varchar(255),
    age    integer,
    locked boolean
);
```

- Entonces, si guardamos un **objeto con valores null**, la entidad se guardará **sin ningún error**:

Valores por defecto en las entidades

- Para crear un valor por defecto directamente en la **definición de la tabla SQL**, podemos usar la **anotación @Column** y trabajar con su parámetro **columnDefinition**:

```
@Entity
public class User {
   @Id
   Long id;

   @Column(columnDefinition = "varchar(255) default 'John Snow'")
   private String name;

   @Column(columnDefinition = "integer default 25")
   private Integer age;

   @Column(columnDefinition = "boolean default false")
   private Boolean locked;
}
```

Valores por defecto en las entidades

- Con este método, el **valor por defecto** estará presente en la **definición de la tabla SQL**:

```
create table user
(
   id     bigint not null constraint user_pkey primary key,
   name   varchar(255) default 'John Snow',
   age    integer      default 35,
   locked boolean      default false
);
```

- Y la entidad se guardará correctamente con los valores por defecto.

Entidades embebidas

- Vamos a ver **cómo podemos asignar** una entidad que contiene propiedades incrustadas a **una sola tabla** de base de datos.
- Para ello, utilizaremos las **anotaciones @Embeddable y @Embedded** proporcionadas por JPA.
- La anotación @Embeddable nos permite definir la clase que queremos inyectar:

```
@Embeddable
public class ContactPerson {
   private String firstName;
   private String lastName;
   private String phone;
   ...
}
```

Entidades embebidas

- La **anotación @Embedded** se usa para **incrustar el tipo en otra entidad**.

```
@Entity
public class Company {

   @Id
   @GeneratedValue
   private Integer id;
   private String name;
   private String address;
   private String phone;

   @Embedded
   private ContactPerson contactPerson;
   ...
}
```

Entidades embebidas

- Sin embargo, todavía tenemos **un problema más**, y es cómo JPA **asignará estos campos** a las **columnas** de la base de datos.

- La **anotación @AttributeOverrides** se encarga de ello:

```
@Embedded
@AttributeOverrides({
  @AttributeOverride( name = "firstName", column = @Column(name = "contact_first_name")),
  @AttributeOverride( name = "lastName", column = @Column(name = "contact_last_name")),
  @AttributeOverride( name = "phone", column = @Column(name = "contact_phone"))
})
private ContactPerson contactPerson;
```

Trabajando con valores nulos

- Vamos a comparar la **anotación @NotNull** de Hibernate con la **anotación @Column(nullable = false)**.
- A primera vista, puede parecer que las **anotaciones @NotNull y @Column(nullable = false)** tienen el mismo propósito y se pueden usar indistintamente. Sin embargo, como pronto veremos, esto **no es del todo cierto**.
- Aunque, cuando se usan en la entidad JPA, ambos **evitan esencialmente almacenar valores nulos** en la base de datos subyacente, **existen diferencias significativas** entre estos dos enfoques.
- Vamos a comparar las restricciones **@NotNull y @Column**(nullable = false).

Trabajando con valores nulos

- La **anotación @NotNull** se define en la **especificación Bean Validation**. Esto significa que su uso **no se limita** solo a las entidades, es decir, también podemos **usar @NotNull** en cualquier otro **bean**.
- Ejemplo:

```
@Entity
public class Item {

   @Id
   @GeneratedValue
   private Long id;

   @NotNull
   private BigDecimal price;
}
```

Trabajando con valores nulos

- Si forzamos el error nuestro sistema lanzará una excepción de tipo **javax.validation.ConstraintViolationException**.
- Es importante dejar claro que Hibernate no ejecutó el insert SQL. En consecuencia, **los datos no válidos no se guardaron en la base de datos**.
- Esto se debe a que el **evento del ciclo de vida** de la entidad **anterior** a la persistencia llevó a cabo la **validación del bean justo antes** de enviar la consulta a la base de datos.

```
create table item (
   id bigint not null,
    price decimal(19,2) not null,
    primary key (id)
)
```

Trabajando con valores nulos

- Sorprendentemente, Hibernate agrega automáticamente la **restricción no nula a la definición** de la columna price. ¿Como es eso posible?
- Resulta que Hibernate **traduce las anotaciones de validación de beans** aplicadas a las entidades en los metadatos del esquema DDL.
- Esto es bastante conveniente y tiene mucho sentido. Si aplicamos **@NotNull a la entidad**, lo más probable es que también queramos que la columna de la base de datos correspondiente no sea nula.
- En cualquier caso, este comportamiento puede desactivarse poniendo la propiedad **hibernate.validator.apply_to_ddl** a false**.**

Trabajando con valores nulos

- La **anotación @Column** se define como parte de la especificación JPA.
- Se utiliza principalmente en la **generación de metadatos del esquema DDL**. Esto significa que si dejamos que Hibernate genere el esquema de la base de datos automáticamente, aplica la restricción de no nulo a la columna de la base de datos en particular.
- Actualicemos nuestra entidad Item con **@Column(nullable = false)** y veamos cómo funciona:

```
@Entity
public class Item {
   ...
   @Column(nullable = false)
   private BigDecimal price;
}
```

Trabajando con valores nulos

- En primer lugar, podemos notar que Hibernate genera la columna price con la **restricción not null**.

```
create table item (
      id bigint not null,
       price decimal(19,2) not null,
      primary key (id)
)
```

- Además, cuando intentamos guardar datos se crear la consulta insert SQL y se ejecuta. Como resultado, **es la base de datos subyacente la que provocó el error**.

Trabajando con valores nulos

- La **anotación @Column(nullable = false)** se usa **solo** para la **generación de DDL de esquema**.
- Hibernate, sin embargo, puede realizar la validación de la entidad contra los posibles valores nulos, incluso si el campo correspondiente está anotado solo con @Column(nullable = false).
- Para activar esta función de Hibernate, debemos establecer explícitamente la propiedad **hibernate.check_nullability** a true.

Trabajando con tipos básicos

- Introducción.
- La anotación @Basic.
- ¿Por qué usamos esta anotación?
- @Basic vs @Column.

Introducción

- JPA admite **varios tipos de datos Java** como campos persistentes de una entidad, a menudo conocidos como **tipos básicos**.

- Un **tipo básico** se asigna directamente a **una columna en la base de datos**. Estos incluyen primitivos de Java y sus clases contenedoras, String, java.math.BigInteger y java.math.BigDecimal, varias clases de fecha y hora disponibles, enumeraciones y cualquier otro tipo que implemente **java.io.Serializable**.

- Hibernate, como cualquier otro proveedor de ORM, mantiene un registro de tipos básicos y lo usa para resolver el **org.hibernate.type.Type** específico de una columna.

Java type	Database type
String (char, char[])	VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)
Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
int, long, float, double, short, byte	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
byte[]	VARBINARY (BINARY, BLOB)
boolean (Boolean)	BOOLEAN (BIT, SMALLINT, INT, NUMBER)
java.util.Date	TIMESTAMP (DATE, DATETIME)
java.sql.Date	DATE (TIMESTAMP, DATETIME)
java.sql.Time	TIME (TIMESTAMP, DATETIME)
java.sql.Timestamp	TIMESTAMP (DATETIME, DATE)
java.util.Calendar	TIMESTAMP (DATETIME, DATE)
java.lang.Enum	NUMERIC (VARCHAR, CHAR)
java.util.Serializable	VARBINARY (BINARY, BLOB)

La anotación @Basic

- Podemos usar la **anotación @Basic** para marcar una propiedad de tipo básico:

```
@Entity
public class Course {

   @Basic
   @Id
   private int id;

   @Basic
   private String name;
   ...
}
```

La anotación @Basic

- En otras palabras, la **anotación @Basic** aplicada sobre un campo o una propiedad significa que es un tipo básico e Hibernate debería usar el mapeo estándar para su persistencia.

- Tenga en cuenta que es **una anotación opcional**. Y así, podemos reescribir nuestra entidad Course como:

```
@Entity
public class Course {
  @Id
   private int id;
   private String name;
   ...
}
```

- Cuando **no especificamos** la anotación @Basic para un atributo de tipo básico, se asume implícitamente y se aplican **los valores por defecto** de esta anotación.

¿Por qué usamos esta anotación?

- La **anotación @Basic** tiene dos atributos, **opcional y fetch**. Echemos un vistazo más de cerca a cada uno.

- El **atributo optional** es un parámetro **booleano** que define si el **campo o la propiedad marcados permiten valores nulos**. Por defecto es **true**. Por lo tanto, si el campo no es un tipo primitivo, se supone que la columna subyacente admite valores null por defecto.

- El **atributo fetch** acepta un miembro del enumerado Fetch, que especifica si el campo o la propiedad marcados deben cargarse de forma diferida o recuperarse lo antes posible. El valor por defecto es **FetchType.EAGER**, pero podemos permitir la carga diferida configurándolo a **FetchType.LAZY**.

- La **carga diferida solo tendrá sentido** cuando tengamos un **objeto serializable** grande mapeado como un tipo básico, ya que en ese caso, el coste de acceso al campo puede ser significativo.

¿Por qué usamos esta anotación?

- Ejemplo: supongamos que **no queremos permitir valores nulos** para el nombre de nuestro curso y que también queremos cargar esa propiedad de forma perezosa.

```
@Entity
public class Course {

   @Id
   private int id;

   @Basic(optional = false, fetch = FetchType.LAZY)
   private String name;
   ...
}
```

@Basic vs @Column

- Veamos las diferencias entre las anotaciones @Basic y @Column:
  - Los **atributos de la anotación @Basic** se aplican a las **entidades JPA**, mientras que los **atributos de @Column** se aplican a las **columnas de la base de datos**.
  - El **atributo opcional de la anotación @Basic** define si el **campo** de entidad puede ser **nulo** o no; por otro lado, el **atributo nullable** de la **anotación @Column** especifica si la **columna** de la base de datos correspondiente puede ser **nula.**
  - Podemos usar @Basic para indicar que un campo debe cargarse de forma diferida.
  - La **anotación @Column** nos permite especificar el nombre de la columna de la base de datos mapeada.

Trabajando con objetos pesados

- JPA nos permite mediante la **anotación @Lob** mapear en la base de datos objetos pesados, como por ejemplo, **imágenes, xml, binarios, cadenas de texto** extensas, json, etc. Cualquier objeto que pueda tener un tamaño muy grande o de longitud indefinida.

- La **anotación @Lob es lo único que se requiere** para indicarle a JPA que ese campo es un objeto pesado y que debe tratarse como tal. Por lo general, se utiliza con los arrays de bytes, ya que permite almacenar cualquier cosa.

- La **anotación @Lob** no tiene **ningún atributo**, por lo que, solo será necesario definirla para que funcione.

- Otro punto importante es que esta anotación creará una **columna de tipo longblob** en mysql y podría variar según el gestor de base de datos utilizados, pero al final siempre será un campo para objetos pesados.

Trabajando con objetos pesados

- Ejemplo:

```
@Entity
@Table(
   name = "EMPLOYEES" ,
   schema = "jpatutorial",
   indexes = {@Index(name = "name_index", columnList = "name",unique = true)}
)
public class Employee {

    @Column(name = "PHOTO" ,nullable = true)
    @Basic(optional = false, fetch = FetchType.EAGER)
    @Lob()
    private byte[] photo;
```

Trabajando con objetos pesados

- Observemos que **@Lob** es mucho más simple de lo que parece, sin embargo, recordemos que **siempre** serán necesarias las anotaciones **@Column y @Basic** para **definir los atributos** de la columna y la estrategia de carga.

- Por lo general, las propiedades **marcadas como @Lob** son cargadas en la creación del objeto y **no siempre es requerido el objeto pesado**, por lo que, es aconsejable utilizar la **anotación @Basic y marcar el atributo como LAZY**.

```
@Entity @Table( name = "EMPLOYEES" ,  schema = "jpatutorial", indexes = {@Index(name = "name_index", columnList =
"name",unique = true)}
)
public class Employee {
   @Column(name = "PHOTO" ,nullable = true)
   @Basic(fetch = FetchType.LAZY)
   @Lob()
   private byte[] photo;
```