Taller de Persistencia con Java

Encuentro 1:

"EL API JPA"

JPA

- El tiempo de desarrollo de soluciones basadas en JDBC es muy alto
  - Brecha entre el modelo relacional y el modelo OO
  - Objetivo: no tener que acoplar el modelo de negocio con lógica de persistencia.
  - Los mecanismos de persistencia deberían ser no – intrusivos con los modelos del dominio
  - Los frameworks ORM que conocemos hoy surgen como una alternativa a los «EJB de entidad» los cuales eran intrusivos dado que debían extender otros componentes.

JPA

- El “Java Persistence API (JPA 2.1)” está definida en la JSR JSR 338.
- Java opera con datos que se encuentran en memoria.
  - La memoria no es infinita para poder almacenar simultáneamente los datos de todos los usuarios.
  - Si se apaga el equipo se pierden los datos.
- Se necesita un mecanismo de persistencia permanente
  - Archivos y serialización → poco eficiente.
  - JDBC para acceso a RDBMS → poco productive
  - EntityBeans → Se usaron hasta J2EE 1.4 y no eran una Buena solución
- Quiero una herramienta que automáticamente guarde los datos en una DB o los lea, cuando los necesite sin que tenga que escribir SQL ➔ la solución de la comunidad fue Hibernate y Java lo estandarizo en JPA

JPA

- JPA define un API para administrar la pesistencia y realizar el mapeo entre el modelo de objetos y el modelo relacional.
- Transforma “automáticamente” y de manera transaccional, cada instancia de un objeto en una fila de la base de datos en la table asignada.
  - Guarda también las relaciones entre objetos mediante el modelo relacional.
  - Cada tabla, en un sentido amplio se corresponde con una clase.
  - Se basa en clases POJO y en almacenar su “estado” en una DB relacional.
  - El estado de un objeto, según se define en la POO, es el valor de todos los atributos que tiene una instancia en un momento dado.

JPA

- Cuando hablamos de mapeo de ~~objetos~~ a bases de datos relacionales, ~~objetos~~ persistentes, o consulta de ~~objetos~~, deberíamos usar el término "**entidad**".
  - Objetos son instancias que solo viven en memoria.
  - Entidades son objetos que temporalmente, y en un lapso breve, se encuentran en memoria, pero persistentemente almacenan su estado en una base de datos relacional.
- Al ser POJO, una entidad es creada, instanciada y usada como cualquier otra clase.
- Para que cualquier clase sea reconocida como entidad tiene que cumplir 2 condiciones:
  - Tener la anotación @Entity
  - Tener un atributo definido como clave primaria (@Id)

JPA

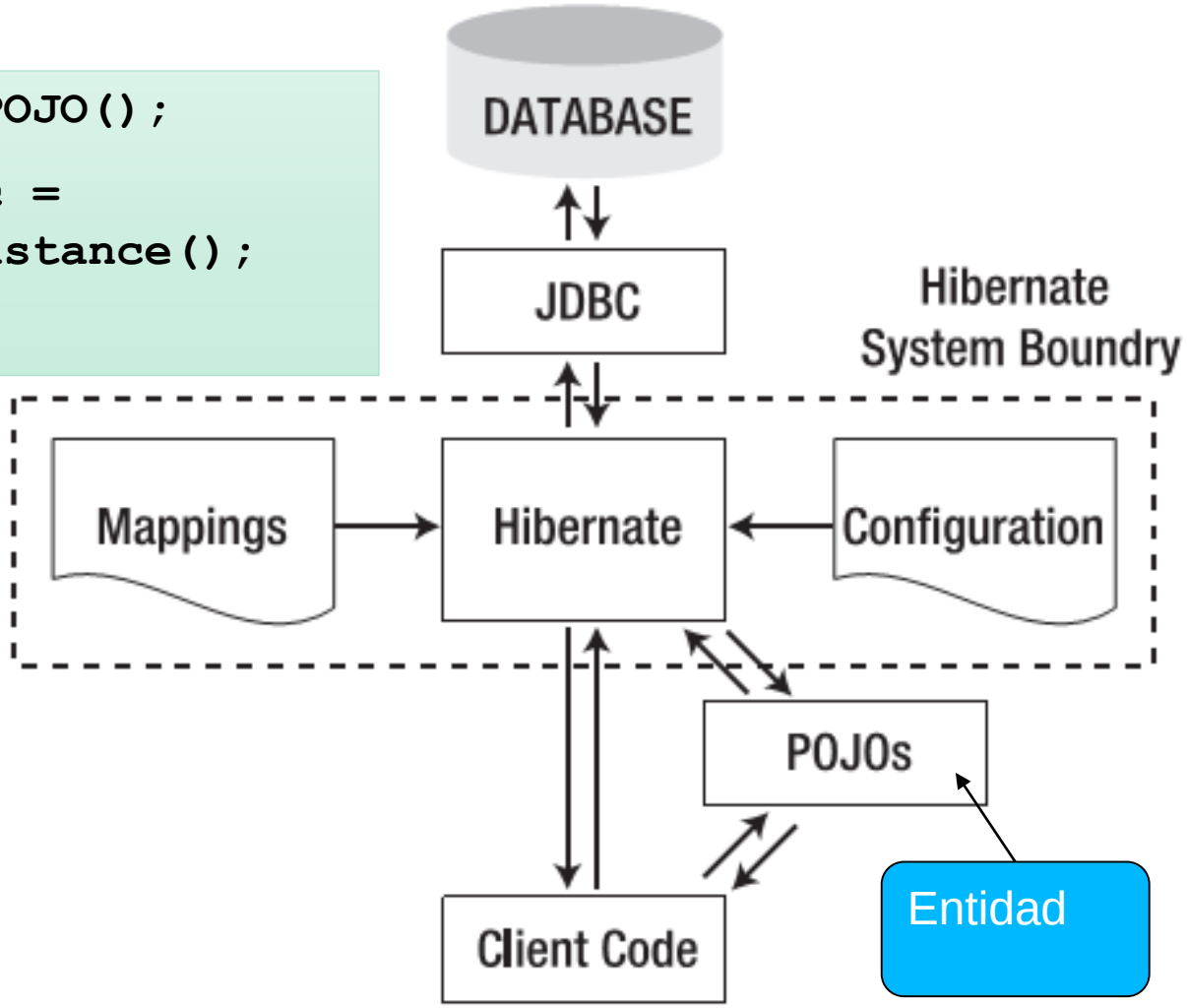
- La principal ventaja desde el punto de vista de desarrollo que nos trae JPA es que nos permite vincular entidades con las bases de datos y trabajar con las entidades con un enfoque orientado a objetos, el cual es mucho más potente y flexible que el de las bases de datos relacionales que nos obligarían a lidiar con tablas, columnas, claves primarias, productos cartesianos y restricciones.
- No obstante todos estos conceptos están subyacentes, pero administrados por nuestro proveedor de persistencia
- Los más populares son:
  - Hibernate
  - EclipseLink (la versión 2.5 es la implementación de referencia)
  - OpenJPA
  - iBatis

JPA – Implementación Hibernate

- Hibernate es un framework iniciado en 2001 por Gavin King, el cual lo ideo como una alternativa flexible a los EJB de Entidad (2.0).
- Objetivo: simplificar las posibilidades de persistencia de aplicaciones empresariales estándares.
- En 2003 el equipo de desarrollo inició la versión Hibernate2 con muchas mejoras, y rápidamente se convirtió en el estándar de facto para el desarrollo de aplicaciones empresariales en JAVA.
- En 2005 Gavin King formó parte del equipo que especificó JPA (estandarizó los mecanismos de persistencia en Java).
- Actualmente Hibernate pertenece a Jboss (RedHat) y está en la versión 4.7

Principio de funcionamiento de Hibernate

```
1. POJO pojo = new POJO();
2. ORMSolution magic =
   ORMSolution.getInstance();
3. magic.save(pojo);
```



Que clases son guardadas automáticamente ? Entidades!

- Una clase **entidad** debe estar anotada con @javax.persistence.Entity (o en el descriptor xml).
- La anotación @javax.persistence.Id debe ser usada para identificar la clave primaria.
- La clase definida como entidad debe asegurar poseer un constructor sin argumentos (por defecto).
  - Puede tener otros, pero es obligatorio el constructor por defecto.
- Una interface o un tipo “enum” no puede ser una entidad.
- La clase definida como entidad **no puede ser final**
- Si debe ser transferida a través de la red, entonces debe implementar la interface Serializable.

Mapeos

- El mapeo de clases y atributos a tablas y columnas, nos permite delegar en frameworks la tarea de administrar la relación entre estos “dos mundos”.
  - De forma transparente podemos trabajar con una visión orientada a objetos “contra la base de datos”.
- El mapeo se puede realizar de dos maneras:
  - Mediante anotaciones (preferido)
  - Mediante archivos XML (útil solo si cambia la **estructura** de la base de datos dependiendo del entorno
- En cualquiera de los dos casos sigue una premisa básica

“Todas las entidades y sus campos se mapean, explícitamente, mediante anotaciones y/o archivos descriptores xml o implícitamente, tomando el mapeo por defecto correspondiente a cada uno”.

Ejemplo de entidad

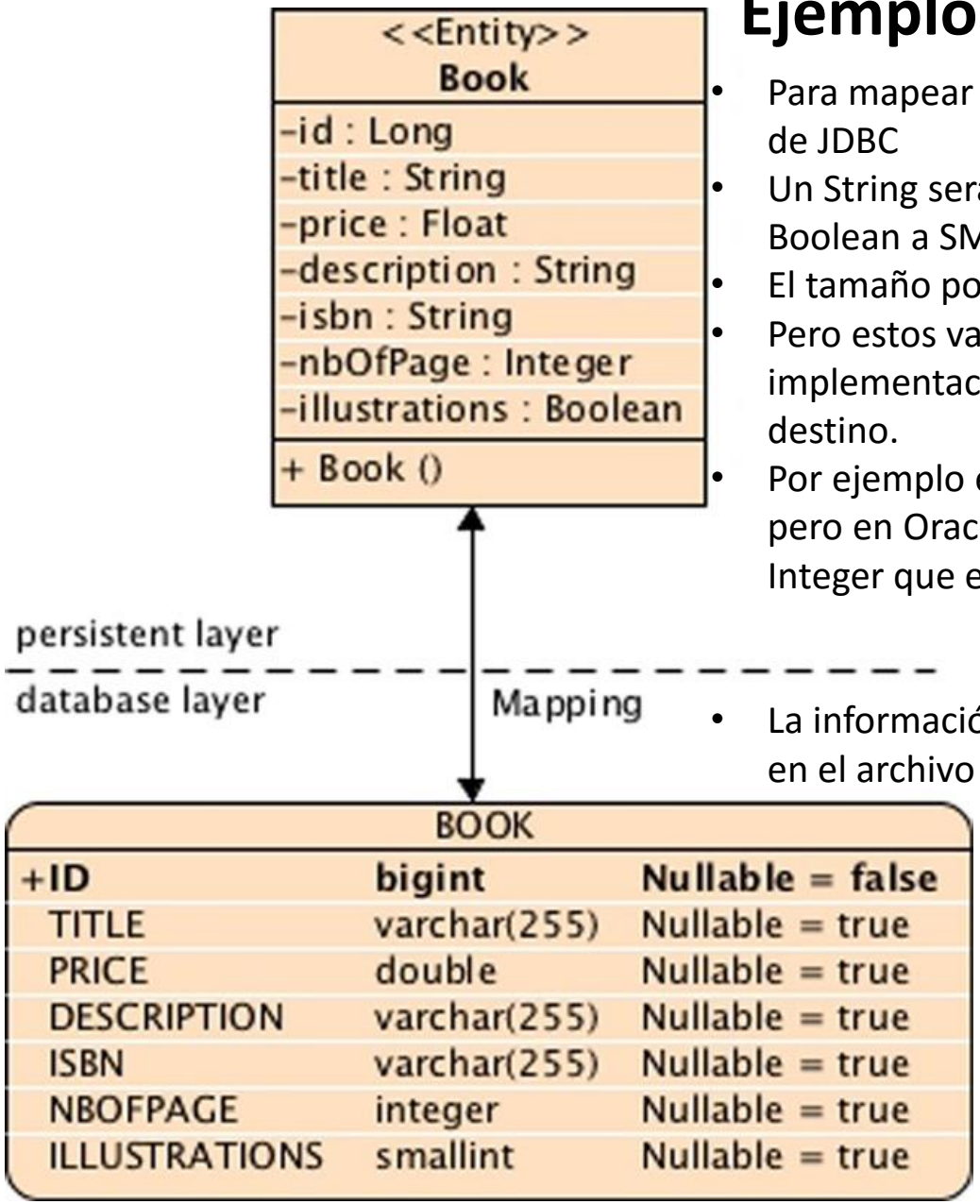
```
1.  @Entity
2.  public class Book {
3.    @Id @GeneratedValue
4.    private Long id;
5.    private String title;
6.    private Float price;
7.    private String description;
8.    private String isbn;
9.    private Integer nbOfPage;
10.   private Boolean illustrations;
11.   public Book() { }
12.   // Getters, setters
```

Es una entidad. No se indica tabla (nombre de tabla por defecto “Book”)

No poseen anotaciones entonces toman los mapeos por defecto (tipo de datos en la DB sale del tipo del atributo y el nombre sale del nombre del atributo).

Principio de “configuration by exception” o “convención sobre configuración”.

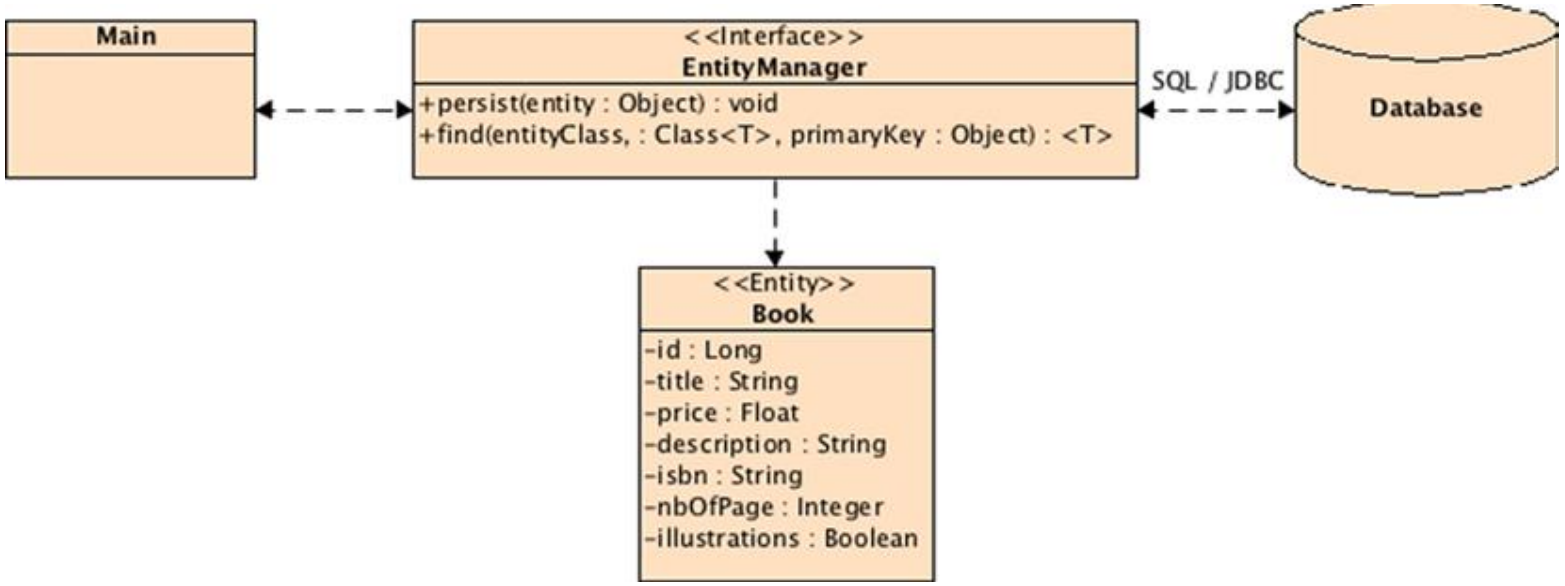
Ejemplo



- Para mapear por defecto las columnas se usan las reglas de JDBC
- Un String será mapeado a VARCHAR, un Long a BIGINT, Boolean a SMALLINT
- El tamaño por defecto para un string es de 255.
- Pero estos valores pueden variar, según la implementación de JPA que usemos, y la base de datos destino.
- Por ejemplo en Derby un String se mapea a VARCHAR pero en Oracle a VARCHAR2 in Oracle. Lo mismo con un Integer que en derby es INTEGER y NUMBER en Oracle.
- La información sobre la base de datos destino se brinda en el archivo persistence.xml.

JPA

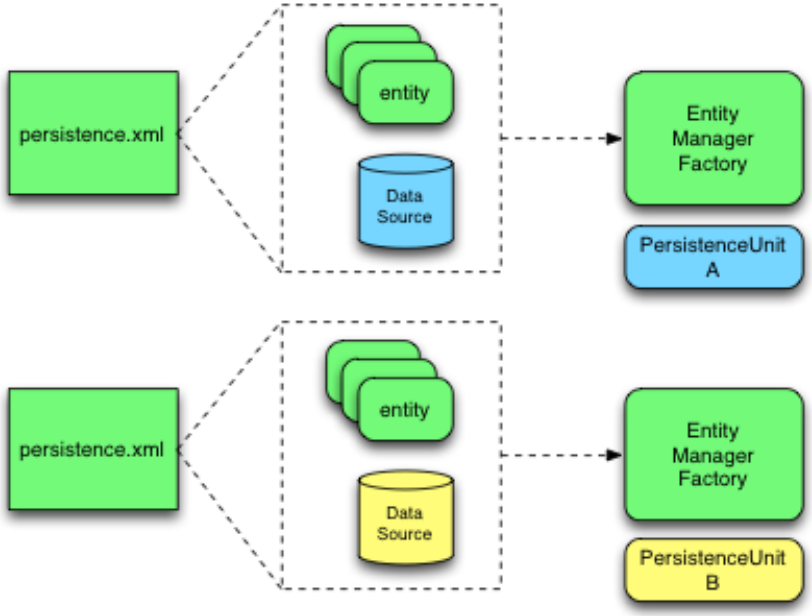
- En JPA el elemento central que administra todo el trabajo entre el modelo de objetos y una base de datos relacional, se llama EntityManager y se encuentra definido por la interface javax.persistence.EntityManager.
  - Lee y escribe en la base de datos, realizando operaciones CRUD, o consultas complejas.

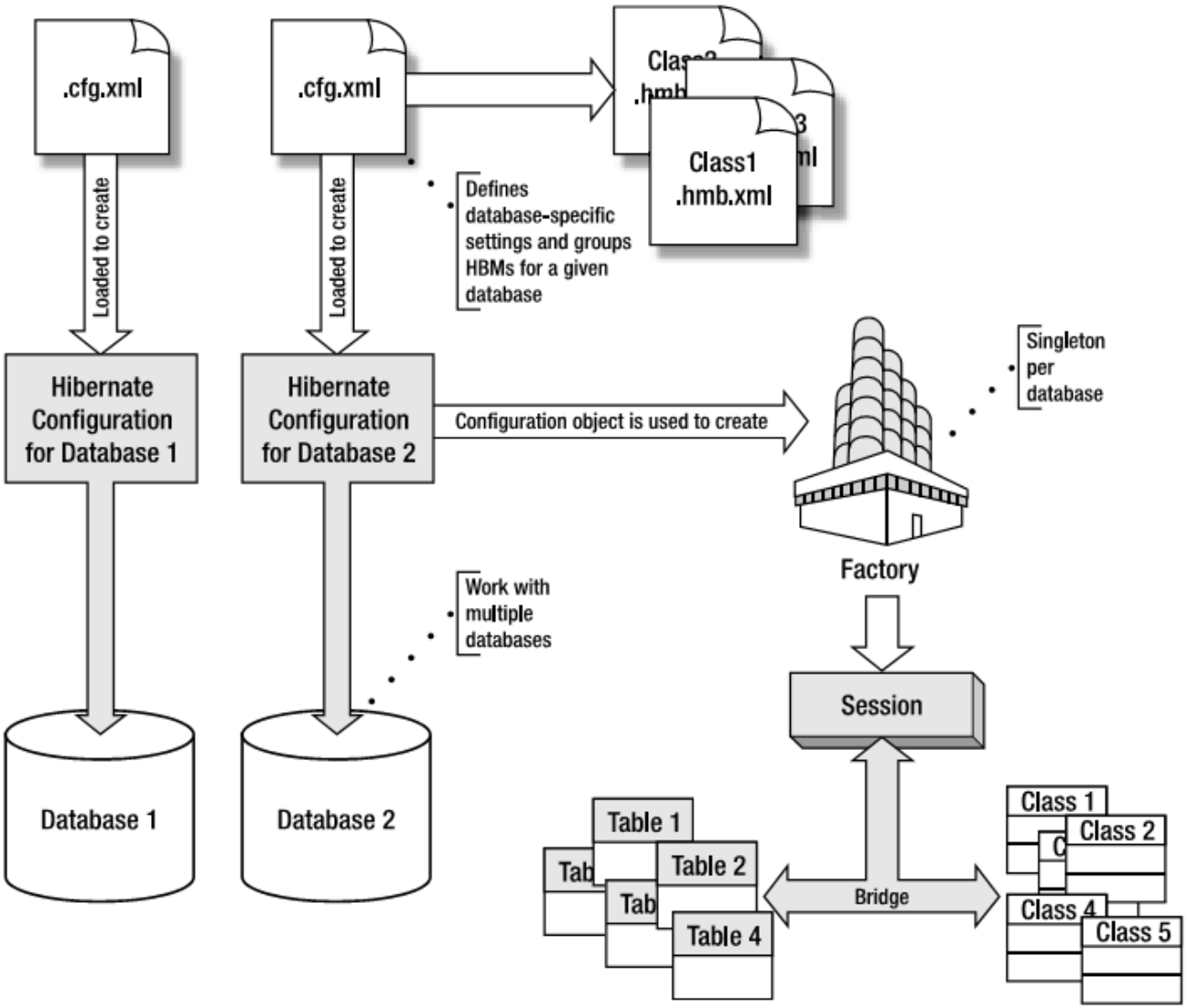


El EntityManager

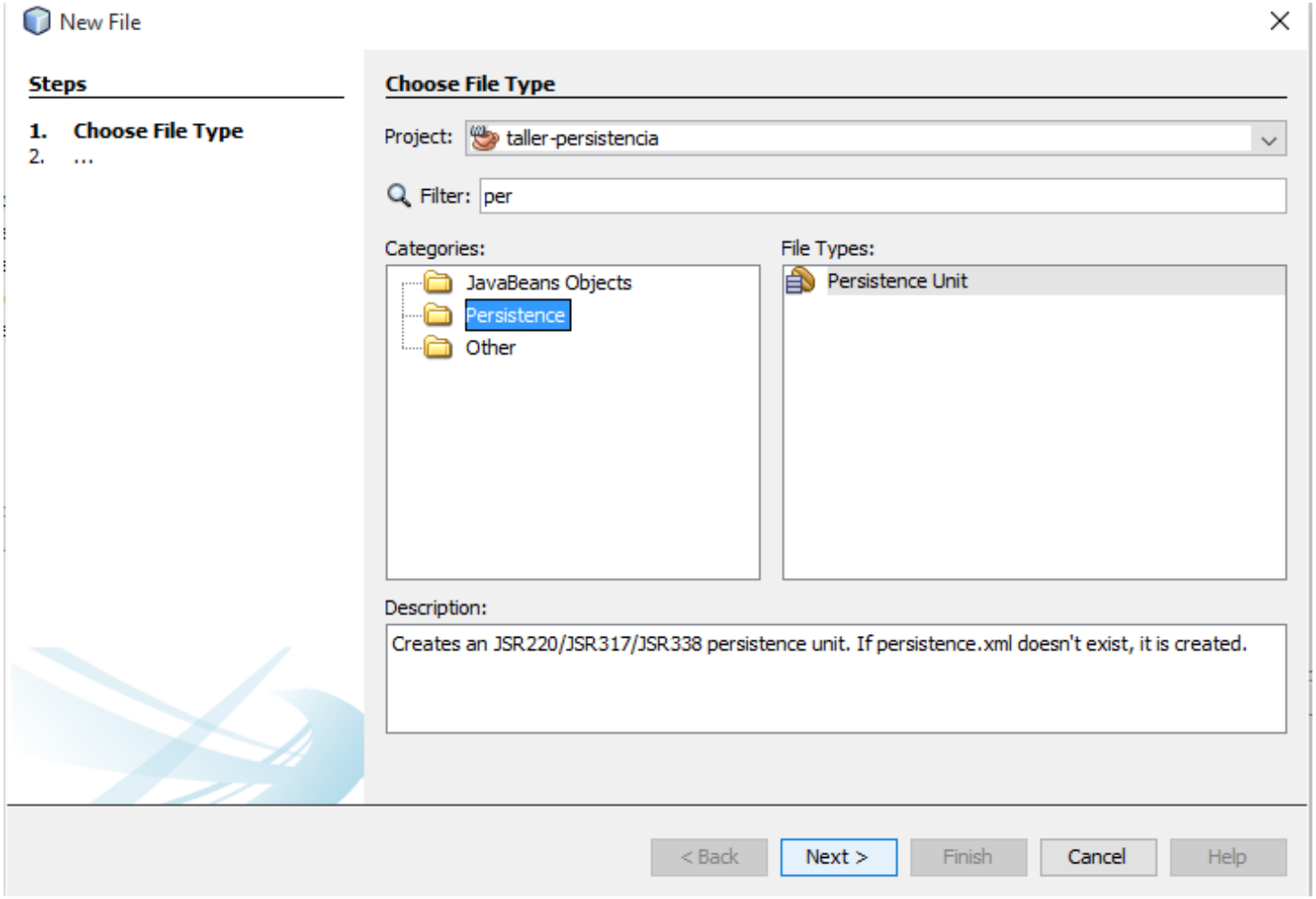
- Para trabajar con el entity manager, necesitamos crear una instancia del mismo.
- Primero debemos obtener el EntityManagerFactory, el cual lee el archivo de configuración apropiado y nos retorna una instancia del EntityManager que maneja una **unidad de persistencia** específica.
- Una unidad de persistencia es un concepto lógico que permite definir como un paquete único conceptos relacionados para administrar la persistencia: como me conecto a la DB, con que driver JDBC, cuales entidades administro, que logs debo dejar, tengo que generar el esquema, etc.

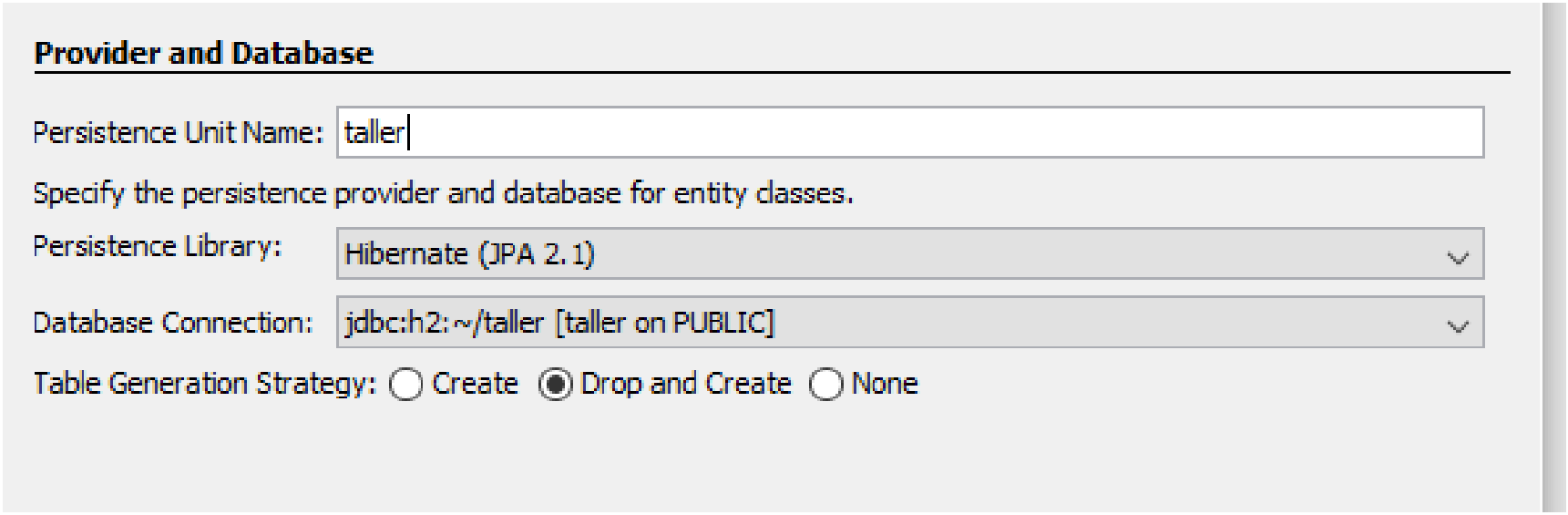
- Un entityManager, es válido para una unidad de persistencia específica. Por ello debemos solicitarlo a un Factory.
- El EntityManagerFactory se encarga de recabar toda esta información y dársela al entity manager, leyéndola a través del archivo "**persistence.xml**".
  - Para poder ser procesado este archivo debe estar en el classpath de la aplicación




.cfg.xml
.cfg.xml
Class1
.hmb.xml
Loaded to create
Loaded to create
Defines database-specific settings and groups HBMs for a given database
Hibernate Configuration for Database 1
Hibernate Configuration for Database 2
Configuration object is used to create
Singleton per database
Factory
Work with multiple databases
Session
Database 1
Database 2
Table 1
Table 2
Table 4
Bridge
Class 1
Class 2
Class 4
Class 5

Crear una unidad de persistencia en el proyecto.




Provider and Database
Persistence Unit Name: taller
Specify the persistence provider and database for entity classes.
Persistence Library: Hibernate (JPA 2.1)
Database Connection: jdbc:h2:~/taller [taller on PUBLIC]
Table Generation Strategy: Create Drop and Create None

Configuración del descriptor “persistence.xml”

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
3.    <persistence-unit name="taller" transaction-type="RESOURCE_LOCAL">
4.     <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
5.      <class>ar.edu.utn.frsf.persistencia.taller.persistencia.modelo.Alumno</class>
6.      <shared-cache-mode>NONE</shared-cache-mode>
7.      <properties>
8.        <property name="hibernate.cache.provider_class"
    value="org.hibernate.cache.NoCacheProvider"/>
9.        <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
10.       <property name="hibernate.show_sql" value="true"/>
11.       <property name="javax.persistence.jdbc.url" value="jdbc:h2:~/taller;FILE_LOCK=NO"/>
```

```
1.    <property name="javax.persistence.jdbc.user" value="taller"/>
2.        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
3.        <property name="javax.persistence.jdbc.password" value="taller"/>
4.        <property name="javax.persistence.schema-generation.database.action"
      value="drop-and-create"/>
5.      </properties>
6.     </persistence-unit>
7.   </persistence>
```

Estandarización de propiedades

- **Conexión a la db**

1. javax.persistence.jdbc.driver: nombre completo del driver
2. javax.persistence.jdbc.url: URL específica del driver
3. javax.persistence.jdbc.user y javax.persistence.jdbc.password

- **Creación de script de base de datos**

1. javax.persistence.ddl-create-script-source : nombre del script que crea la db
2. javax.persistence.ddl-drop-script-source script que borra la db
3. javax.persistence.sql-load-script-source script que carga datos en la db
4. javax.persistence.schema-generation.database.action que hacer con la db en cada despliegu(none, create, drop-and-create, drop)

Schema Generation

- javax.persistence.schemageneration.database.action
  - Especifica que tiene que hacer el proveedor de persistencia en cuanto a la generación del esquema. none (defecto), create, dropand-create, drop
- javax.persistence.schemageneration.scripts.action
  - Especifica que scripts serán generados por el proveedor de persistencia. Un script será generado solamente si el "target" es especificado.
  - Valores posibles: ”none, create, dropand-create, drop”
- javax.persistence.schemageneration.create-source, javax.persistence.schemageneration.drop-source
  - Especifica si la creación o borrado de los elementos de la base de datos, se hace en base a los metadatos, a los scripts pasados, primero en base a metadatos y luego se completa son un script o viceversa.
  - Valores posibles: “metadata, script,metadata-then-script,script-then-metadata”

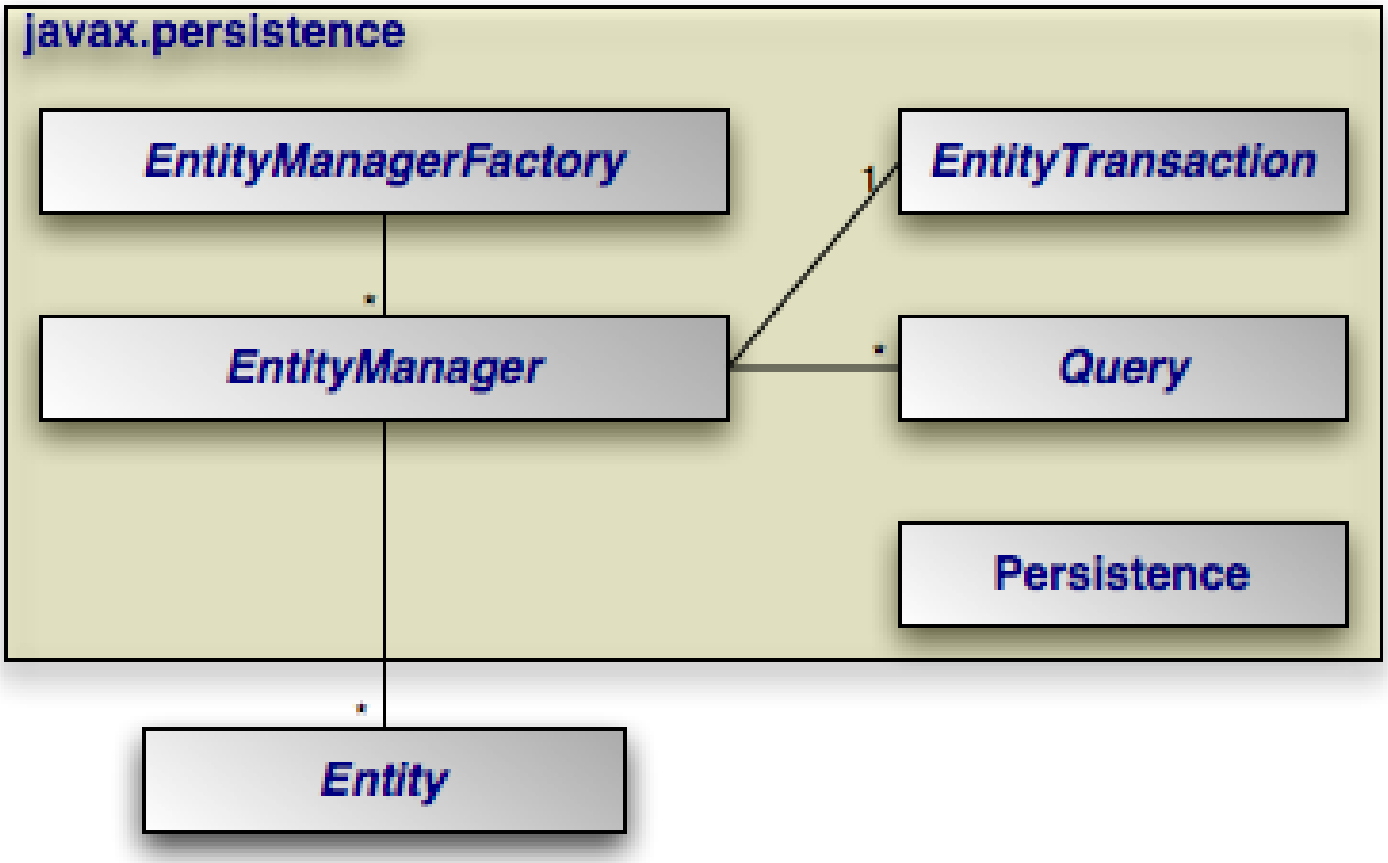
Schema Generation

- Si se especifica "script" la ubicación se debe indicar usando las propiedades javax.persistence.schema-generation.create-script-source o javax.persistence.schema-generation.drop-script-source . La ubicación es relativa a la raíz de la unidad de persistencia.
- javax.persistence.schemageneration.create-databaseschemas
  - Indica si es necesario crear el esquema además de los objetos. (true, false)
- javax.persistence.schema-generation.scripts.create-target,
- javax.persistence.schema-generation.scripts.drop-target
- Indica el archivo donde se crearan los scripts
  - <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
  - <property name="javax.persistence.schema-generation.scripts.action" value="drop-and-create"/>
  - <property name="javax.persistence.schema-generation.scripts.create-target" value="create.ddl"/>
  - <property name="javax.persistence.schema-generation.scripts.drop-target" value="drop.ddl"/>

Como usar JPA

- Obtener una instancia de EntityManagerFactory
  - Obtener solo 1. Cada vez que obtenemos una instancia del EntityManagerFactory se analizan todas las entidades, se validan las tablas y se crea lo que se necesita.
- Una vez que tenenmos el EntityManagerFactory, solicitamos el objeto EntityManager
- Para trabajar con modificación de datos, al entityManager se le solicita acceso a la transacción y se la incializa con “begin()”
- Realizar las operaciones, por ejemplo “persist” y cerrar la transacción (commit) y el entityManager (close)

Obtener un EntityManager



El entityManager se obtiene a través de un entityManagerFactory.

Ejemplo de uso

```
1.  public class Main {
2.  public static void main(String[] args) {
3.  Book book = new Book("H2G2", “Guia 1", 12.5F,"1-84023-742-2", 354, false);
4.  EntityManagerFactory emf = Persistence.createEntityManagerFactory("chapter04PU");
5.  EntityManager em = emf.createEntityManager();
6.  EntityTransaction tx = em.getTransaction();
7.  tx.begin();
8.  em.persist(book);
9.  tx.commit();
10. book = em.find(Book.class,1);
11. em.close();
12. emf.close();
13. }
14. }
```

crear una instancia de la entidad

Obtener un EntityManagerFactory e iniciar una transaccion

persistir en la db

Hacer una busqueda

cerrar el entity manager y el factory

Interactuamos con la base de datos sin escribir SQL!!

Ejercicio

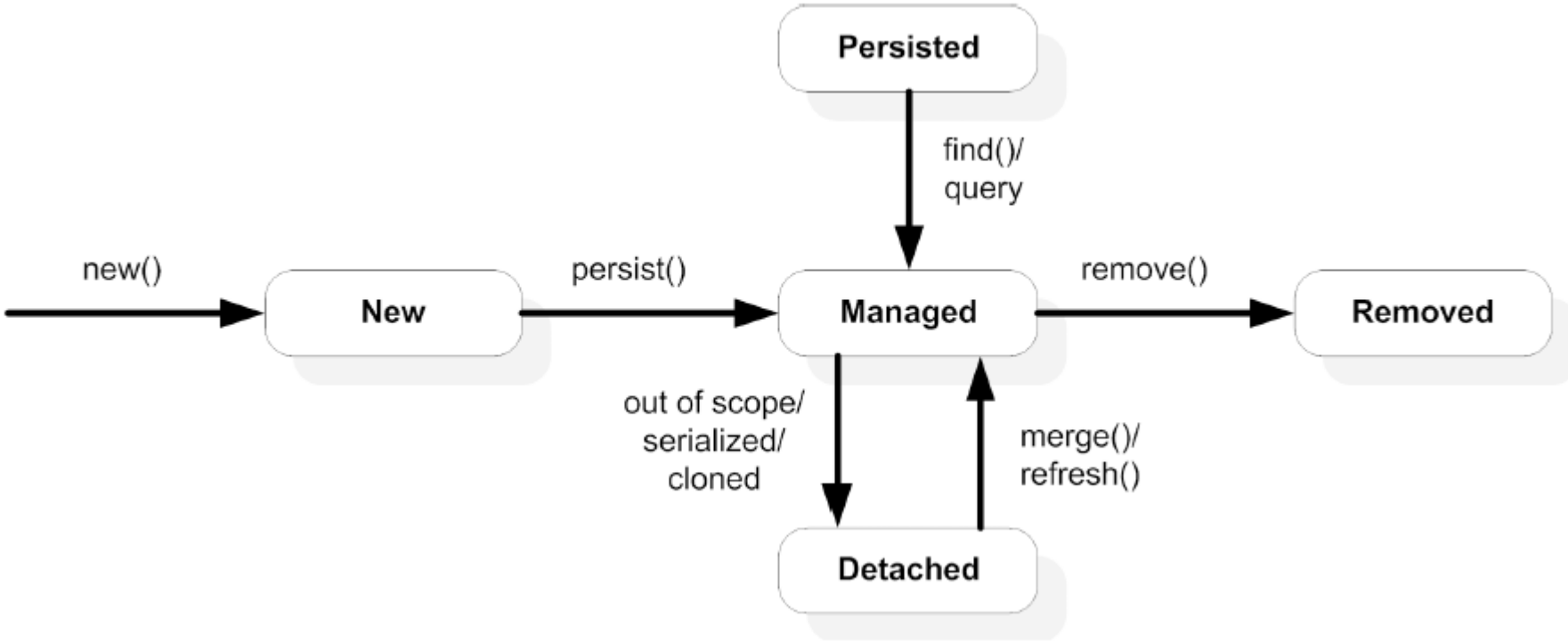
- Crear en el paquete ar.edu.utn.frsf.persistencia.taller.persistencia.dao.útil la clase MiEntityManager que tenga el siguiente código

```
1.  public class MiEntityManager {
2.      private static EntityManagerFactory _entityManagerFactory = null;
3.          private static void inicializar(){
4.          if(_entityManagerFactory==null)
5.          _entityManagerFactory = Persistence.createEntityManagerFactory("taller");
6.      }
7.      public static EntityManager get(){
8.          inicializar();
9.          return _entityManagerFactory.createEntityManager();
10.     }
11.     }
```

Ejercicio

- Implementar la clase “AlumnoDaoJDBC”
  - El método crear
  - El método listarTodos.
    - List<Alumno> result = em.createQuery("SELECT a FROM Alumno a").getResultList();
- Para acceder al EntityManager, utilizar la clase anteriormente definida.
- Cambie el vínculo de AlumnoServiceImpl con el nuevo DAO.

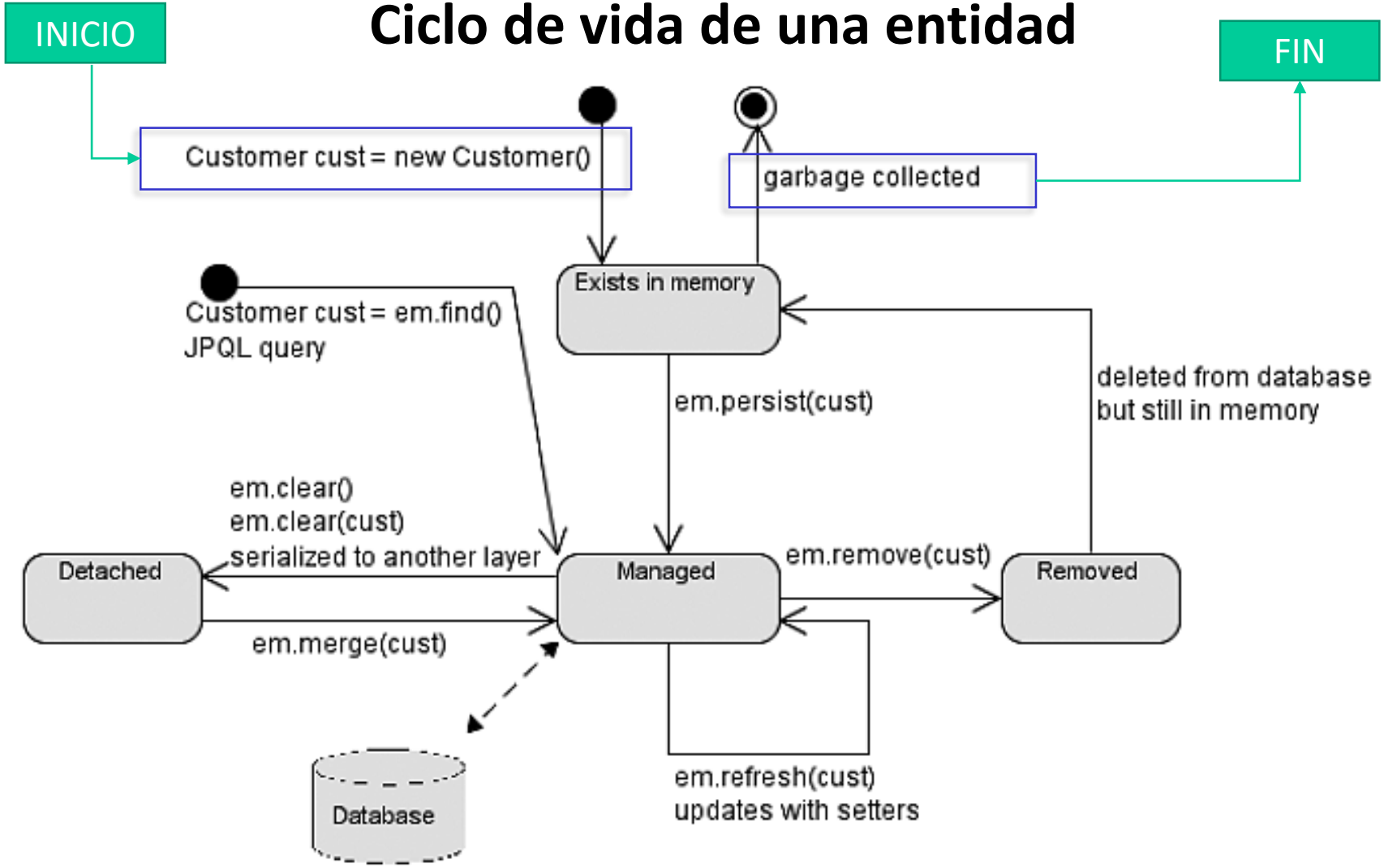
Ciclo de vida de una entidad



Ciclo de vida de la entidad

- Una entidad puede estar en alguno de los siguientes cuatro estados
  - **Nueva:** La entidad esta creada en memoria (como una instancia de la clase persistentes) pero aún no a sido asociada ni con la base de datos ni con el contexto de persistencia. En este caso, la entidad aún esta desvinculada del *EntityManager*.
  - **Gestionada (managed):** La entidad tiene una identidad persistente en la base de datos y esta asociada actualmente con un contexto de persistencia. Una entidad nueva pasa al estado gestionado, luego que el método EntityManager.persist() se halla invocado pasando como parámetro una referencia a la entidad.
  - **Desvinculada (detached):** La entidad tiene una identidad persistente en la base de datos pero no esta asociada a un contexto de persistencia.
  - **Eliminada (removed):** La entidad esta actualmente asociada a un contexto de persistencia pero ha sido marcada para ser eliminada en la base de datos.

Ciclo de vida de una entidad



El EntityManager

- http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html
- Principales métodos del ciclo de vida:
  - void persist(Object entity): hace a un objeto persistente, es decir lo comienza a administrar y pasa del estado transient al estado persistente.
  - <T> T merge(T entity): combina el estado del obejto pasado como parámetro con el del objeto que administra y retorna el objeto persistente! (el que pasamos por parámetro no es persistente)
  - <T> T find(Class<T> entityClass, Object primaryKey): retorna una instancia de un objeto persistente buscada por si PK
  - void remove(Object entity): quita una instancia del estado administrado por lo que cuando la transacción termine la eliminará de la base de datos.

El EntityManager

- Métodos que controlan la sincronización con la DB
  - void refresh(Object entity): vuelve a cargar un objeto de la DB
  - void flush(): fuerza bajar en la DB lo que se encuentra en memoria
  - void setFlushMode(FlushModeType flushMode): especifica cuando realizar la operación de escritura en la DB

    FlushModeType.AUTO (Default): se hace antes de cualquier consulta .

    FlushModeType COMMIT: cuando cierra la transacción
- Métodos para consulta
  - Query createNamedQuery(String name)
  - Query createNativeQuery(String sqlString)
  - Query createQuery(String qlString)

El EntityManager

- EntityManagerFactory getEntityManagerFactory();
- Si quiero obtener la implementación real
  - Object getDelegate();
- Consultar si administra una entidad
  - boolean contains(Object entity);

Ejercicio

- Implementar la clase “AlumnoDaoJDBC”
  - Implementar el resto de los métodos en esta clase.
  - Tenga en cuenta que las consultas las puede resolver de la siguiente manera
  - “SELECT a FROM Alumno a WHERE nombre LIKE ‘%:P1%’”
  - Y luego de creada la query utilizar el método setParameter(“P1”,var);

Otras anotaciones

```
1. @Entity
2. @Table(name="EMP", schema="HR")

3. public class Employee {
4.     @Id
5.     @GeneratedValue(strategy=GenerationType.AUTO)
6.     private int id;
7.    @Column(name="FULL_NAME")
8.    private String name;
9.    private long salary;
   1. ...
10.    public String toString(){ . . . }

11.}
```

Entidades

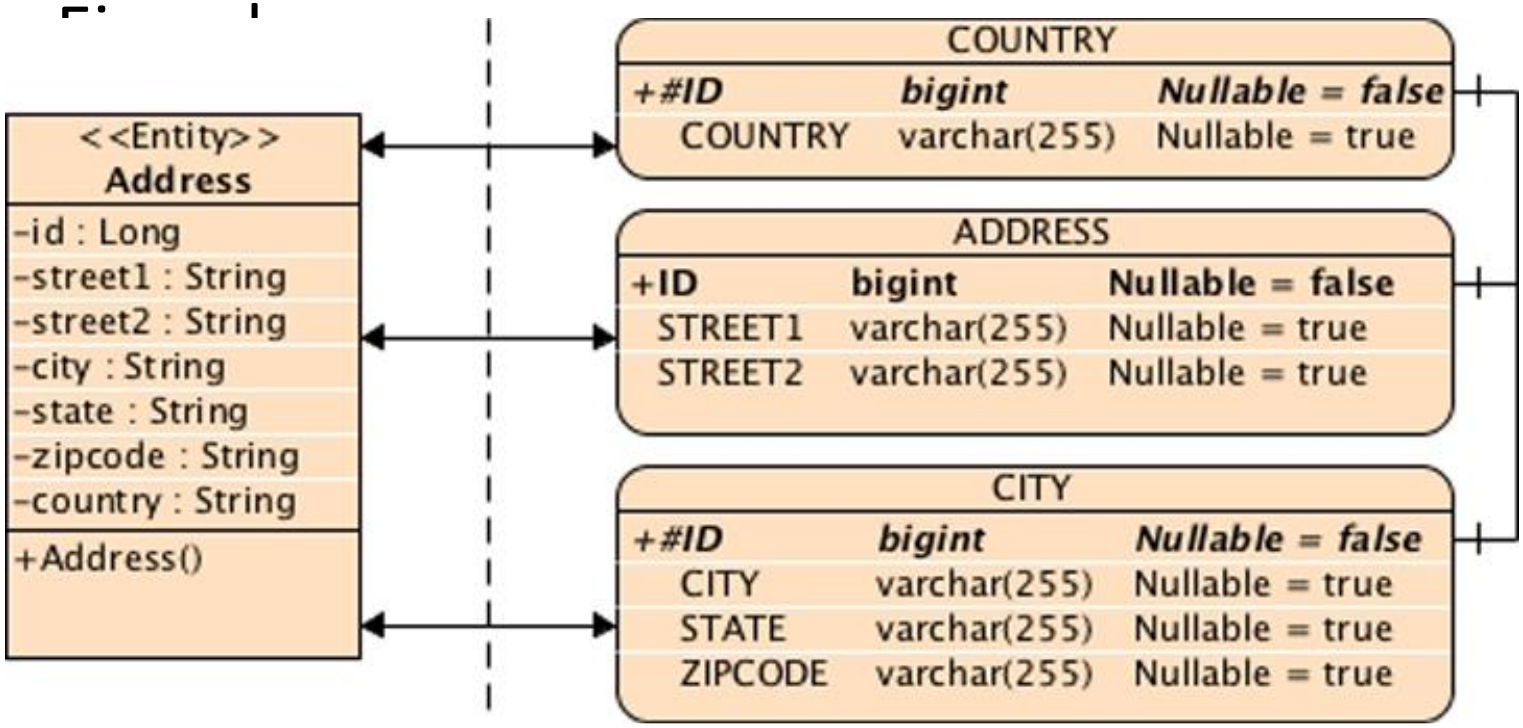
- Principales Anotaciones:
  - ***@javax.persistence.Table*** especifica el nombre de la tabla donde la entidad es almacenada. Por defecto es el nombre de la entidad. Y por defecto el nomber de la entidad es el nombre de la clase.
  - ***@javax.persistence.Column*** puede ser utilizada en métodos o en atributos y define el nombre de la columna de la tabla donde es almacenado el atributo. Por defecto es igual al nombre del campo de la tabla.
  - La anotación @Table puede definir una anotación @UniqueConstraint para indicar una restricción de unicidad

```
1. @Entity
2. @Table( name="EMPLOYEE", uniqueConstraints=
   @UniqueConstraint(columnNames={"EMP_ID", "EMP_NAME"})
   )
3. public class Employee { ... }
```

Entidades

- Principales Anotaciones:
  - ***@javax.persistence.Id*** especifica que propiedad corresponde a clave primaria y ***@javax.persistence.GeneratedValue*** especifica el método que se debe utilizar para generar el valor de la clave primaria para una nueva entidad.

SecondaryTable



SecondaryTable

```
1. @Entity
2. @SecondaryTables({@SecondaryTable(name = "city"),
3. @SecondaryTable(name = "country") })
4. public class Address {
5. @Id private Long id;
6. private String street1;
7. private String street2;
8. @Column(table = "city") private String city;
9. @Column(table = "city") private String state;
10. @Column(table = "city") private String zipcode;
11. @Column(table = "country") private String country;
12. }
```

SecondaryTable

- Cuando se usan tablas secundarias se tiene que tener en cuenta que si bien el modelo queda más “limpio” y normalizado, puede tener implicancias negativas en términos de performance.
- Cada vez que se accede a una entidad mapeada a más de una tabla, como sucede en el caso de tablas secundarias, el proveedor de persistencia necesita realizar múltiples accesos para “unirlas”.
- No obstante son una opción válida cuando tenemos atributos de gran tamaño, como Blobs, que queremos aislar en de una tabla.

@ForeignKey

- La anotación @ForeignKey permite que cuando definamos la relación entre 2 entidades, indiquemos a nivel de bases de datos cual es la clave foránea a crear.
  - Se aplica para JoinColumn(s), MapKeyJoinColumn(s), y PrimaryKeyJoinColumn(s):

```
1. @Entity public class Employee {
2. @Id private int id;
3. private String name;
4. @ManyToOne
5. @JoinColumn(foreignKey=@ForeignKey(
6. foreignKeyDefinition="FOREIGN KEY (MANAGER_ID) REFERENCES
   MANAGER"))
7. private Manager manager; . . .
8. }
```

- En este ejemplo, se crea la FK MANAGER_ID
- foreignKeyDefinition es esepecífico de la base de datos.

@ForeignKey

- Podemos sino simplemente definir cual es el nombre de la FK a usar para la relación

```
1. Entity @Table(name = "state")
2. public class State {
3. @Id @Column(name = "id") private int id;
4. @Column(name = "name") private String name;
5.  @ManyToOne
   @ForeignKey(name="FK_COUNTRY")
6. private Country country;
7. }
```

@ID

- La anotación @Id indica a un atributo como identificador único en el contexto de persistencia.
- Pueden ser atributos del siguiente tipo
  - Tipos primitivos de java: byte, int, short, long, char y sus correspondientes Wrappers Byte, Integer, Short, Long, Character
  - Arreglos de primitivos o wrapper : int[], Integer[], etc.
  - Strings, números , y fechas : java.lang.String, java.math.BigInteger, java.util.Date,
  - java.sql.Date

La anotación @ID – *Estrategias de generación*

- La clave primaria puede ser generada automáticamente por la aplicación o por el proveedor. Si deseamos esto último debemos usar @GeneratedValue que puede tomar 4 posibles valores
- **public enum GenerationType { TABLE, SEQUENCE, IDENTITY, AUTO };**
  - AUTO: permite que proveedor de persistencia defina la mejor forma de generar las claves primarias. (defecto)
  - IDENTITY: usa tipo de datos auto-numérico definido en muchas bases de datos.
  - TABLE: declara que se utiliza una tabla definida por el usuario desde donde se generan los valores para las claves primarias.

    Para utilizar esta estrategia previamente se debe definir la tabla utilizando la anotación *@TableGenerator* la cuál puede ser aplicada sobre una clase, un método o un campo de la clave primaria.
  - SEQUENCE: mecanismo empleado por Oracle para generar claves primarias.

    Este tipo de generador es declarado utilizando @SequenceGenerator.
- Si no especificamos @GeneratedValue, la aplicación los debe generar!!

Ejemplo de estrategia Table

- Es la estrategia mas flexible y portable.
- Ejemplo:

  @Id GeneratedValue(strategy=GenerationType.TABLE)

  private int id;
- En este caso no se le asigna ninguna tabla, por lo que el gestor de persistencia seleccionará una si la opción que permite generar esquemas está activada, caso contrario es necesario definir cual es la tabla que un DBA ya creo o producirá un error.
- Solución emplear la anotación : **@TableGenerator**

La anotación @ID – *Estrategia Sequence*

```
1. package javax.persistence;
2. @Target({METHOD, TYPE, FIELD}) @Retention(RUNTIME)
3. public @interface SequenceGenerator
4. {
5. String name( );
6. String sequenceName( )
7. default "";
8. int initialValue( ) default 1;
9. int allocationSize( ) default 50;
10.}
```

1. Ejemplo
2. @SequenceGenerator(name="Emp_Gen", sequenceName="Emp_Seq")
3. @Id @GeneratedValue(generator="Emp_Gen")
4. private int getId;

La anotación @ID – *Estrategias de generación*

- Estrategia SEQUENCE:
  - El atributo name( ) especifica el nombre por el cual será referenciado esta secuencia en la anotación @Id.
  - El atributo sequenceName( ) define el nombre de la secuencia en la base de datos
  - El atributo initialValue( ) es el primer valor que será utilizado primer clave primaria y allocationSize( ) especifica en cuanto se debe incrementar este valor .

Ejemplo de la estrategia IDENTITY Y AUTO

- **Identity**: emplea el autonumérico que utilizan múltiples motores de base de datos. Un inconveniente que tiene es que el identificador no está disponible hasta después de insertar un dato en una tabla. No necesita de un "generator".

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

- **Auto**:

```
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
```

Tener en cuenta que para algunas estrategias es necesario crear tablas o secuencias entonces el usuario de la base de datos debe tener dichos privilegios.
Esta estrategia en realidad es útil para la etapa de desarrollo o para prototipos.

Anotaciones para atributos: La anotación @Column

- Describe como un campo particular es mapeado a una columna especifica en una tabla.

```
public @interface Column
{
    String  name( ) default "";
    boolean unique( ) default false;
    boolean nullable( ) default true;
    boolean insertable( ) default true;
    boolean updatable( ) default true;
    String  columnDefinition( ) default "";
    String  table( ) default "";
    int     length( ) default 255;
    int     precision( ) default 0;
    int     scale( ) default 0;
}
```

La anotación @Column

- *name()* especifica el nombre de la columna, Por defecto es el nombre de la propiedad de la clase.
- *table()* es utilizado cuando se realiza mapeo de una clase, en más de una tabla.
- *unique()* y *nullable()* define las restricciones que se implementaran sobre la columna, *unique* agrega una clave única sobre la columna, y *nullable* permite asignar valores nulos.
- Mediante *insertable() y updatable()* se indica si la columna debe ser incluida en los SQL de INSERT o UPDATE generados por el *Entity Manager*.
- Podemos aplicar el API de BeanValidation sobre las entidades. No obstante estos chequeos se realizan sobre memoria.
- Por ejemplo @NotNull chequea que un valor esté en memoria y @Column(nullable = false) es usado para indicarle a la base de datos que no acepte persistir datos que no cumplan con esa restricción.
- Ambas validaciones pueden coexistir en un mismo atributo.

Ejemplo

```
1.  @Entity
2.  public class Book {
3.  @Id @GeneratedValue(strategy = GenerationType.AUTO)
4.  private Long id;
5.  @Column(name = "book_title", nullable = false, updatable =
    false)
6.  private String title;
7.  private Float price;
8.  @Column(length = 2000)
9.  private String description;
10. private String isbn;
11. @Column(name = "nb_of_page", nullable = false)
12. private Integer nbOfPage;
13. private Boolean illustrations;
14. // Constructors, getters, setters
15. }
```

SQL GENERADO

```
1.  create table BOOK (
2.  ID BIGINT not null,
3.  BOOK_TITLE VARCHAR(255) not null,
4.  PRICE DOUBLE(52, 0),
5.  DESCRIPTION VARCHAR(2000),
6.  ISBN VARCHAR(255),
7.  NB_OF_PAGE INTEGER not null
8.  ILLUSTRATIONS SMALLINT,
9.  primary key (ID)
10. );
```

- *columnDefinition()* permite especificar el DDL que se debe utilizar para crear el campo en la tabla.
- Para valores numéricos se pueden definir *scale*( ) y *precision*( ) para informar acerca de cantidad de posiciones enteras y decimales a utilizar.