FRAMEWORK

Spring Data
ORM: Hibernate + JPA

Módulo V

harriague+asociados

Emilio Rey | María Agustina Aliaga

INCLUIT
empowering your business

Object Relational Mapping (ORM)

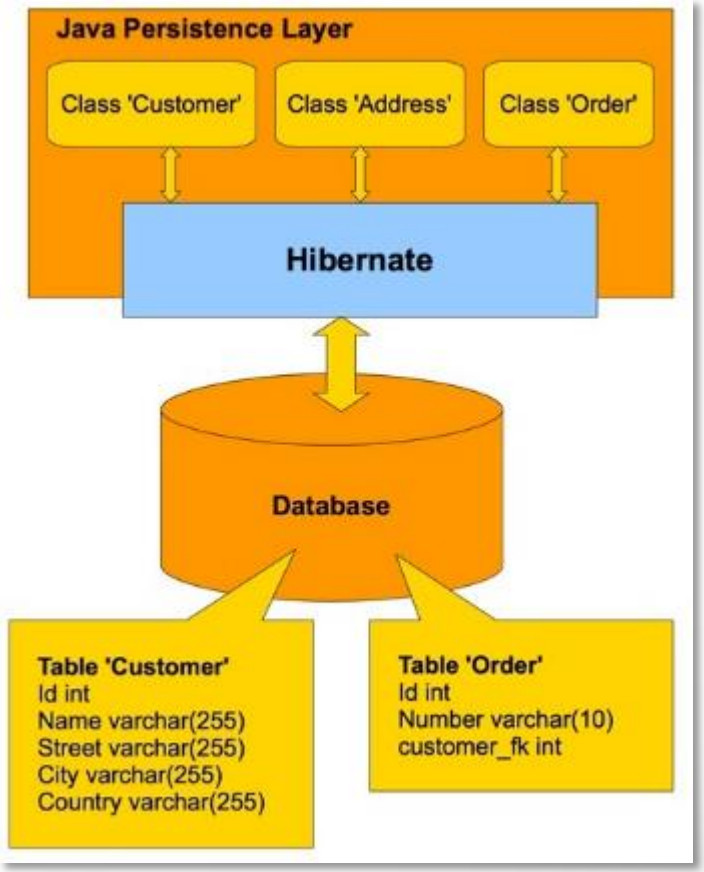
Se trata de **mapear** o **convertir datos representados en forma de objetos a datos representados relacionalmente** y viceversa.

Su **función** principal es **asociar un objeto a sus datos en la BD,** posibilitando escribir las clases de persistencia utilizando OO.

Al mapear debemos tener en consideración el tipo de datos y sus relaciones.

Hay muchos **ORM** para Java: EJB, JDO, **JPA**.

Mientras que estos son especificaciones, **Hibernate**, es una **implementación**.

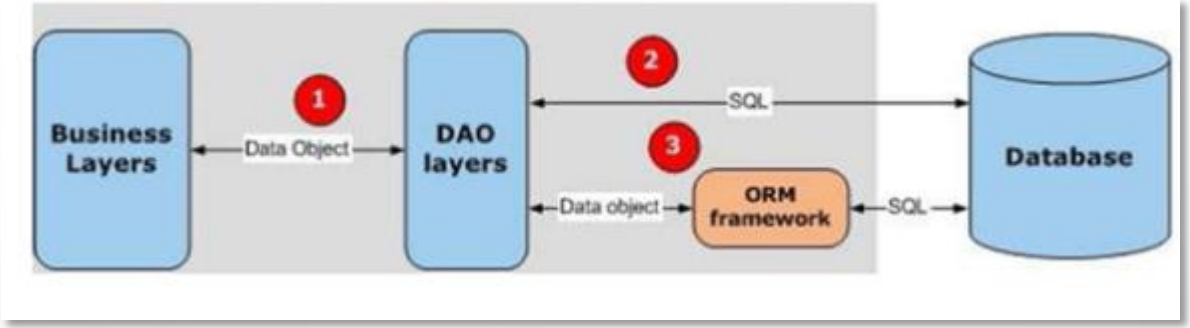


Por qué ORM

Ventajas de ORM:

- Permite al **código de negocios acceder a objetos en vez de tablas** de BD.
- **Oculta los detalles de bajo nivel** de las consultas SQL.
- No es necesario preocuparse por la **implementación** de la BD.
- Las **entidades pueden basarse en conceptos de negocios** en vez de en la estructura de la BD.
- **Manejo de transacciones y generación automática de claves**.

En conclusión, desarrollo más rápido y ágil de la aplicación.



Qué es JPA

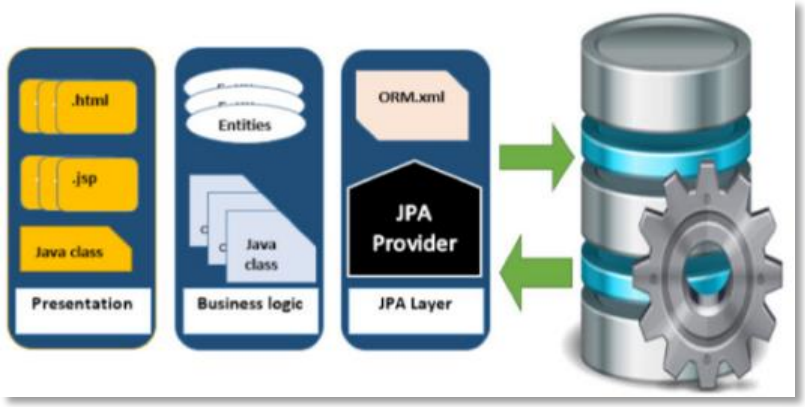
Java Persistence API es una **colección de clases y métodos** que almacenan de forma persistente grandes cantidades de datos.

JPA **busca solucionar la problemática planteada al intentar traducir un modelo orientado a objetos a un modelo relacional**.

Permite **almacenar entidades del negocio como entidades relacionales**.

Los problemas que surgen se refieren a la granularidad, herencia, identidad, asociaciones y navegación de los datos.

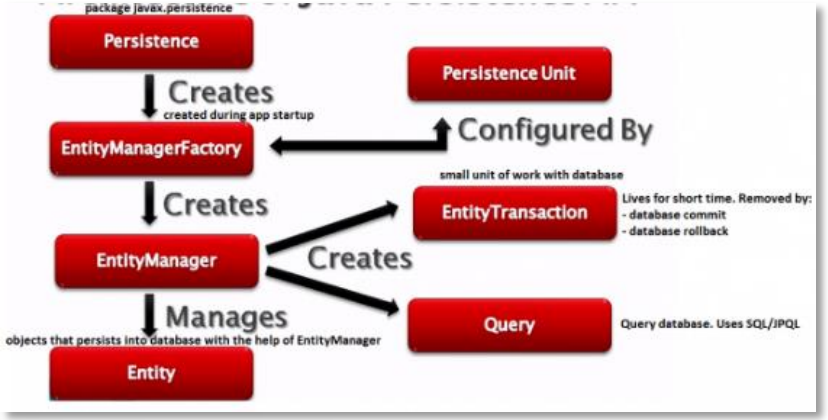
El desarrollador ahora solo **debe implementar el framework del proveedor de JPA** que permite una interacción sencilla con la instancia de la BD.



JPA es una API open source y existen diferentes **proveedores** que lo **implementan**, siendo utilizado en productos como por ejemplo **Hibernate**, Spring Data JPA, etc.

Arquitectura de JPA

El gráfico muestra las **clases** e **interfaces** que conforman el **núcleo** de JPA.



Paquete	Funcionalidad
EntityManagerFactory	Es una **clase** Factory que **crea y gestiona múltiples instancias del Entity Manager.**
EntityManager	Es una **interfaz** que **gestiona las operaciones de persistencia en los objetos**. Funciona como una **fábrica de la instancia Query**.
EntityTransaction	Por cada EntityManager hay una **EntityTransaction que mantiene las operaciones.**
Query	**Interfaz implementada por cada vendor JPA** para realizar **consultas** a la BD.

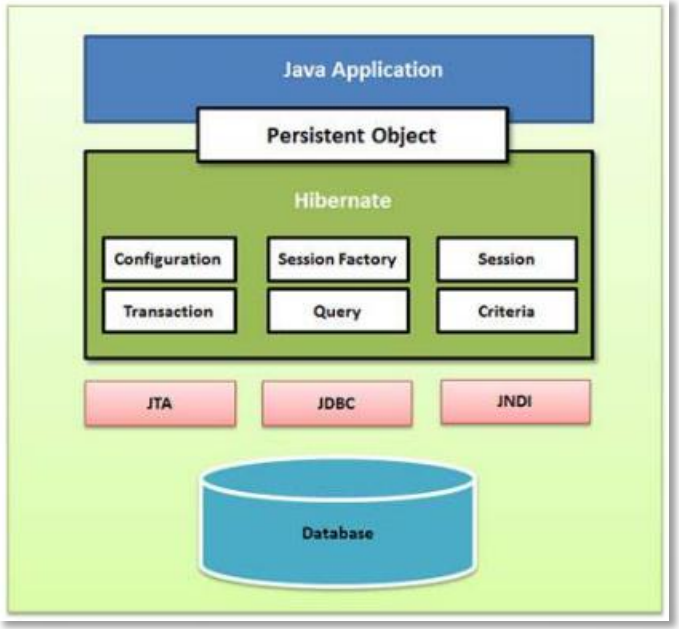
HIBERNATE

Hibernate es un s**ervicio ORM de persistencia y consultas**.

Mapea las clases Java en tablas de BD, y provee mecanismos para consultar datos.

El **mapeo** lo hace a través de una **configuración .xml o de anotaciones,** si es necesario un cambio en la BD, solo deberá cambiarse el archivo de configuración.

Arquitectura de HIBERNATE



Spring + Hibernate con JPA

Lo primero es **además de Spring JDBC, añadir la dependencia de Hibernate core y de spring ORM en el pom.xml.** Desde antes tendremos spring JDBC.

```
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-jdbc</artifactId>
 <version>4.3.0.RELEASE</version>
</dependency>

<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-orm</artifactId>
 <version>4.3.0.RELEASE</version>
</dependency>

<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-core</artifactId>
 <version>4.3.6.Final</version>
</dependency>
```

Luego será necesario **declarar una SessionFactory**.

Session provee la **funcionalidad** básica de **acceso a datos** que satisfará todas las necesidades de persistencia de los DAO.

El **SessionFactory** de Hibernate es una **implementación de los FactoryBean** de Spring.

Es posible **añadir** la **configuración** de Hibernate tanto en **xml** como en una clase Java agregando **anotaciones**.

Configuración XML vs. Anotaciones

```xml
<?xml version='1.0' encoding='utf-8'?>
<!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>
    <!-- Database connection settings -->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/stockdb</property>
    <property name="connection.username">root</property>
    <property name="connection.password">secret</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">true</property>

    <mapping class="net.codejava.hibernate.Category"/>
    <mapping class="net.codejava.hibernate.Product"/>

  </session-factory>
</hibernate-configuration>
```

Session Factory es configurado con tres propiedades:

- **dataSource**: referencia a un bean dataSource.
- **mapping**: Lista los recursos donde se encuentra el mapeo.
- **hibernateProperties**: se configura por ejemplo el uso de HSQLDialect.

```java
@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.jackrutorial.config" })
@PropertySource(value = { "classpath:config.properties" })
public class HibernateConfig {

 @Autowired
 private Environment environment;

 @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] { "com.jackrutorial.model" });
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
     }

 @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));
        dataSource.setPassword(environment.getRequiredProperty("jdbc.password"));
        return dataSource;
    }

 private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));
        properties.put("hibernate.show_sql", environment.getRequiredProperty("hibernate.show_sql"));
        properties.put("hibernate.format_sql", environment.getRequiredProperty("hibernate.format_sql"));
        return properties;
    }
```

Creando un Entity Bean

Un **EntityBean** es un **POJO** (Plain Old Java Objects) es decir un **objeto de negocio que debe ser persistido por una BD relacional**.

Las anotaciones son provistas por el paquete JPA javax.persistence.

Anotaciones JPA:

@**GeneratedValue**: Indica a Hibernate que utilice una columna de id autogenerada y elige cual forma se utilizará.

@**Table**: Indica que este POJO será mapeado a una tabla con el nombre de customer.

@**Column**: indica a Hibernate que los atributos serán mapeados como una columna de la BD.

```
@Entity
@Table(name="customer")
public class Customer {

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

 @Column(name = "firstname")
 private String firstname;

 @Column(name = "lastname")
 private String lastname;

 @Column(name = "gender")
 private String gender;

 @Column(name = "address")
 private String address;

 public int getId() {
  return id;
 }

 public void setId(int id) {
  this.id = id;
 }

 public String getFirstname() {
  return firstname;
 }

 public void setFirstname(String firstname) {
  this.firstname = firstname;
 }
```

DAO

Tal como vimos en el módulo anterior, debemos **crear una interfaz DAO genérica, que sea implementada** por interfaces DAOs específicas.

Por ejemplo, tendremos una implementación para Spring JDBC Template, y otra para Hibernate.

```
public interface CustomerDao {

 public List listAllCustomers();

 public void saveOrUpdate(Customer customer);

 public Customer findCustomerById(int id);

 public void deleteCustomer(int id);

}
```

CustomerDao va a implementar una interfaz de DAO genérica con los métodos CRUD.

Luego, crearemos **CustomerDaoHibernateImpl.java**, que será la implementación de la interfaz para Hibernate con JPA que anotamos como @Repository.

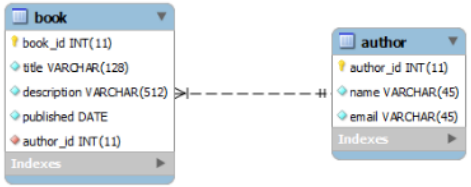
```
@Repository
public class CustomerDaoImpl implements CustomerDao {

 @Autowired
 private SessionFactory sessionFactory;

 protected Session getSession(){
  return sessionFactory.getCurrentSession();
 }

 @SuppressWarnings("unchecked")
 public List listAllCustomers() {
  Criteria criteria = getSession().createCriteria(Customer.class);
  return (List) criteria.list();
 }

 public void saveOrUpdate(Customer customer) {
  getSession().saveOrUpdate(customer);
 }

 public Customer findCustomerById(int id) {
  Customer customer = (Customer) getSession().get(Customer.class, id);
  return customer;
 }

 public void deleteCustomer(int id) {
  Customer customer = (Customer) getSession().get(Customer.class, id);
  getSession().delete(customer);
 }
}
```

@Repository:

- Puede ser escaneado por el component-scan.
- Requiere de manejo de excepciones específico en tiempo de ejecución.
- Podemos especificar el nombre con que va a reconocerse en el contenedor.

@OneToOne Mapping

Utilizaremos las **anotaciones JPA** para implementar una **asociación unidireccional** con una FK.




```java
@Entity
@Table(name = "AUTHOR")
public class Author {
    private long id;
    private String name;
    private String email;

    public Author() {
    }

    public Author(String name, String email) {
        this.name = name;
        this.email = email;
    }

    @Id
    @Column(name = "AUTHOR_ID")
    @GeneratedValue
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
```

```java
@Entity
@Table(name = "BOOK")
public class Book {
    private long id;
    private String title;
    private String description;
    private Date publishedDate;

    private Author author;

    public Book() {
    }

    @Id
    @Column(name = "BOOK_ID")
    @GeneratedValue
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Temporal(TemporalType.DATE)
    @Column(name = "PUBLISHED")
    public Date getPublishedDate() {
        return publishedDate;
    }

    public void setPublishedDate(Date publishedDate) {
        this.publishedDate = publishedDate;
    }

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "AUTHOR_ID")
    public Author getAuthor() {
        return author;
    }
```

Ejemplo @OneToOne

- @**Entity**: Es necesario para cada clase del modelo.
- @**Table**: Mapea la clase con la tabla correspondiente en la BD.
- @**Column**: Mapea el campo con la columna correspondiente. Si es omitido, Hibernate inferirá el nombre de la columna y el tipo por los getters y setters.
- @**Id** y @**GeneratedValue**: Son utilizados en conjunto para un campo que es mapeado a una PK. En este caso se indica que los valores serán autogenerados.
- @**Temporal**: Debe ser utilizado con un campo del tipo java.util.Date para especificar el tipo que será asignado en SQL.
- @**OneToOne** y @**JoinColumn**: son utilizados en conjunto para especificar una asociación uno a uno y la columna de unión.

Otra manera de hacerlo es configurar este **mapeo en .xml**, aunque la forma más utilizada es hacerlo a través de anotaciones.

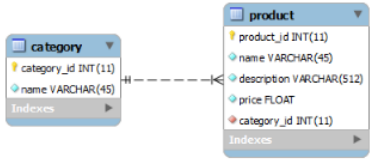
Author.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="net.codejava.hibernate">
    <class name="Author" table="AUTHOR">
        <id name="id" column="AUTHOR_ID">
            <generator class="native"/>
        </id>
        <property name="name" column="NAME" />
        <property name="email" column="EMAIL" />
    </class>
</hibernate-mapping>
```

Book.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="net.codejava.hibernate">
    <class name="Book" table="BOOK">
        <id name="id" column="BOOK_ID">
            <generator class="native" />
        </id>
        <property name="title" type="string" column="TITLE" />
        <property name="description" type="string" column="DESCRIPTION" />
        <property name="publishedDate" type="date" column="PUBLISHED" />

        <many-to-one name="Author" class="net.codejava.hibernate.Author"
            column="author_id" unique="true" not-null="true"
            cascade="all" />

    </class>
</hibernate-mapping>
```

@ManyToOne / @OneToMany




```java
@Entity
@Table(name = "CATEGORY")
public class Category {

    private long id;
    private String name;

    private Set<Product> products;

    public Category() {
    }

    public Category(String name) {
        this.name = name;
    }

    @Id
    @Column(name = "CATEGORY_ID")
    @GeneratedValue
    public long getId() {
        return id;
    }

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL)
    public Set<Product> getProducts() {
        return products;
    }

    // other getters and setters...
}
```

```java
@Entity
@Table(name = "PRODUCT")
public class Product {
    private long id;
    private String name;
    private String description;
    private float price;

    private Category category;

    public Product() {
    }

    public Product(String name, String description, float price,
            Category category) {
        this.name = name;
        this.description = description;
        this.price = price;
        this.category = category;
    }

    @Id
    @Column(name = "PRODUCT_ID")
    @GeneratedValue
    public long getId() {
        return id;
    }

    @ManyToOne
    @JoinColumn(name = "CATEGORY_ID")
    public Category getCategory() {
        return category;
    }

    // other getters and setters...
}
```

Ejemplos @OneToMany / @ManyToOne

En **Category.java** el atributo **mappedBy** es necesario, porque especifica que **la asociación es mapeada desde este lado.**

El atributo **cascade** asegura que Hibernate va a **guardar/actualizar el Set de productos cuando se actualice la categoría**.

```
private Set<Product> products;

@OneToMany(mappedBy = "category", cascade = CascadeType.ALL)
public Set<Product> getProducts() {
    return products;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="net.codejava.hibernate">
    <class name="Category" table="CATEGORY">
        <id name="id" column="CATEGORY_ID">
            <generator class="native"/>
        </id>
        <property name="name" column="NAME" />

        <set name="products" inverse="true" cascade="all">
            <key column="CATEGORY_ID" not-null="true" />
            <one-to-many class="Product"/>
        </set>
    </class>
</hibernate-mapping>
```

En **Products.java** la anotación @**JoinColumn** especifica la **columna** en la que se **realizará la unión**, y es necesario.

```
private Category category;

@ManyToOne
@JoinColumn(name = "CATEGORY_ID")
public Category getCategory() {
    return category;
}
```

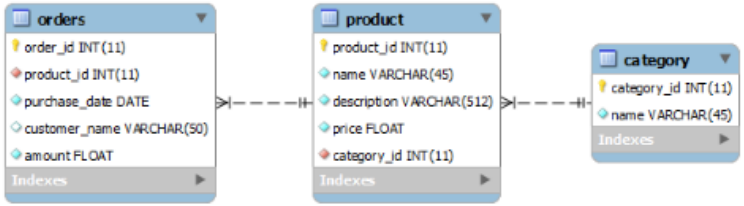
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="net.codejava.hibernate">
    <class name="Product" table="PRODUCT">
        <id name="id" column="PRODUCT_ID">
            <generator class="native"/>
        </id>
        <property name="name" column="NAME" />
        <property name="description" column="DESCRIPTION" />
        <property name="price" column="PRICE" type="float" />

        <many-to-one name="category" class="Category"
            column="CATEGORY_ID" not-null="true"/>
    </class>
</hibernate-mapping>
```

Hibernate Query Language

Hibernate provee su propio **lenguaje de consultas**.

Es parecido a SQL en su sintaxis pero es totalmente **orientado a objetos** (utiliza los nombres de las **clases** y **atributos**).

Al ejemplo anterior, sumamos la clase orders. El DER quedaría de la siguiente manera:



Y la nueva clase Order.java sería así:

```java
@Entity
@Table(name = "ORDERS")
public class Order {
    private int id;
    private String customerName;
    private Date purchaseDate;
    private float amount;
    private Product product;

    @Id
    @Column(name = "ORDER_ID")
    @GeneratedValue
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    @Column(name = "CUSTOMER_NAME")
    public String getCustomerName() {
        return customerName;
    }

    @Column(name = "PURCHASE_DATE")
    @Temporal(TemporalType.DATE)
    public Date getPurchaseDate() {
        return purchaseDate;
    }


    @ManyToOne
    @JoinColumn(name = "PRODUCT_ID")
    public Product getProduct() {
        return product;
    }

    // other getters and setters
}
```

Ejemplos de HQL

Select que trae todos los objetos Category.

```
String hql = "from Category";
Query query = session.createQuery(hql);
List<Category> listCategories = query.list();

for (Category aCategory : listCategories) {
    System.out.println(aCategory.getName());
}
```

En HQL es posible **omitir** la palabra **SELECT**, y utilizar solamente FROM.

El método **list()** retorna una lista de Objetos de la clase Category. Si no encuentra ningún resultado, retornará la lista sin ningún elemento.

Búsqueda por **nombre.**

```
String hql = "from Product where category.name = 'Computer'";
Query query = session.createQuery(hql);
List<Product> listProducts = query.list();

for (Product aProduct : listProducts) {
    System.out.println(aProduct.getName());
}
```

En este caso Hibernate **genera automáticamente la consulta JOIN entre las tablas Product y Category**, por lo que no es necesario utilizar la palabra JOIN.

Ejemplos de HQL

Select utilizando un parámetro.

```
String hql = "from Product where description like :keyword";

String keyword = "New";
Query query = session.createQuery(hql);
query.setParameter("keyword", "%" + keyword + "%");

List<Product> listProducts = query.list();

for (Product aProduct : listProducts) {
    System.out.println(aProduct.getName());
}
```

Busca todos los Products en cuya **descripción** figure la **keyword** especificada, en este caso «New».

Insert … Select

```
String hql = "insert into Category (id, name)"
        + " select id, name from OldCategory";

Query query = session.createQuery(hql);

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
    System.out.println(rowsAffected + "(s) were inserted");
}
```

HQL no tiene soporte para un INSERT regular, por lo que se utiliza el **INSERT … SELECT.**

Se ejecuta una consulta que inserta todas las filas de la tabla Category en la tabla OldCategory.

Ejemplos de HQL

Update

```
String hql = "update Product set price = :price where id = :id";

Query query = session.createQuery(hql);
query.setParameter("price", 488.0f);
query.setParameter("id", 43l);

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
    System.out.println("Updated " + rowsAffected + " rows.");
}
```

Ejecuta una consulta que actualiza el precio de un producto con id determinado.

Delete

```
String hql = "delete from OldCategory where id = :catId";

Query query = session.createQuery(hql);
query.setParameter("catId", new Long(1));

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
    System.out.println("Deleted " + rowsAffected + " rows.");
}
```