



Grado en Ingeniería Informática

APLICACIONES DE BASES DE DATOS

TEMA 5

Sección 2. JPA: Introducción y Entidades

Docentes:

Raúl Marticorena

Jesús Maudes

Mario Martínez



Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	3
3. JPA: INTRODUCCIÓN.....	3
3.1 Concepto.....	3
3.2 Características.....	4
3.3 Arquitectura.....	4
4. ENTIDADES.....	5
4.1 Acceso a atributos y propiedades.....	7
4.2 Tipos Java.....	9
4.3 Carga de valores.....	10
4.4 Clases embebidas.....	10
4.5 Claves primarias: simples vs. compuestas.....	11
4.6 Claves primarias: generación de valores únicos.....	14
4.7 Tipos enumerados.....	15
4.8 Tipos temporales.....	16
4.9 Datos transitorios.....	17
4.10 Anotaciones de columna.....	17
5. RESUMEN.....	18
6. GLOSARIO.....	18
7. BIBLIOGRAFÍA.....	18
8. RECURSOS.....	18



1. Introducción

En la siguiente sección se introduce el concepto de la API de persistencia para Java – Java Persistence API (JPA desde ahora) – y sus propiedades básicas, mostrando la arquitectura general con la que se trabajará en el resto de la asignatura. A continuación se revisan los conceptos básicos para el adecuado ajuste (*mapping*) entre clases/objetos (en LPOO) y tablas/registros (en BD relacionales) a través del concepto de entidades y el uso de anotaciones Java.

2. Objetivos

- Conocer JPA, sus propiedades básicas y arquitectura.
- Conocer las reglas básicas de mapeo entre clases (entidades) y tablas, junto con sus correspondientes objetos y registros respectivamente.

3. JPA: Introducción

A lo largo de los años siempre existió un gran problema con el desajuste entre el mundo de los objetos y el mundo de las bases de datos relacionales. Este problema se ha denominado tradicionalmente en la literatura como *impedance mismatch*. Básicamente, no encajan dichos modelos: la conversión de objetos a filas en las tablas y viceversa no es directa, ni trivial.

Existiendo ciertas soluciones comerciales de herramientas ORM (Object-Relational Mapping) en el mercado, se intentó la definición de una API general para distintos proveedores de *frameworks* de persistencia (e.g. en Java: Hibernate, EclipseLink, TopLink, etc.)¹.

3.1 Concepto

Como solución al problema planteado surge el concepto de JPA o API de persistencia para Java. Su versión 1.0 nace dentro de la especificación de EJB 3.0, a la sombra del concepto fallido de los *Entity JavaBeans*, para posteriormente tomar cuerpo de especificación independiente en versiones 2.0 y posteriores.

JPA se apoya desde su primera versión en el concepto de Plain Old Java Object (POJO). Como su propio nombre indica, son clases Java muy simples ("planas", con atributos y métodos *getter/setter* inicialmente aunque no exclusivamente) a las que se adornan con algún aditamento sintáctico del lenguaje (anotaciones) para que pasen a tomar un rol diferente en la aplicación (entidades en el caso de JPA).

JPA 2.0 añade alguna funcional adicional como aumento en las capacidades de mapeo, mayor flexibilidad en el modo de acceso al estado de las entidades, mejora de la API de consulta (JP QL) y creación de criterios de consulta dinámicos. En la versión 2.1 se añaden nuevas mejoras, pero de menor calado, como conversores, mejoras en las consultas, etc. La especificación 2.2 está pendiente todavía de revisión, en una versión de mantenimiento, y por lo tanto de aquí en adelante tomaremos como referencia final sólo la versión 2.1.

¹ En cierto sentido, una evolución similar a JDBC como API de acceso a bases de datos, con una cierta independencia del proveedor particular de SGBD.



Básicamente se trata de utilizar el concepto de **metadatos** para dirigir el desarrollo: evolucionando desde soluciones con descriptores en ficheros XML externos (solución heredada de la especificación previa de EJB) al uso intensivo de anotaciones (desde que fueran incluidas en Java 1.5).

3.2 Características

Para trabajar con JPA es necesario trabajar con POJOs. Estas clases, que siguen el modelo de JavaBeans (modelo muy antiguo en Java) con atributos y sus correspondientes métodos `get` y `set`, tienen una correspondencia con una entidad persistente y con su correspondiente tabla en la base de datos. Es decir, sus datos en memoria se deben recuperar y guardar en la base de datos de una manera transparente.

Para ello **no** se hace uso de la API JDBC directamente. Se añaden anotaciones a la clase que ayudan al *framework* de persistencia a realizar las correspondientes operaciones SQL.

Inicialmente es desconcertante trabajar con los POJOs, por la ausencia inicial de código JDBC y SQL embebido, aunque como veremos posteriormente, esta aparente simplicidad se complica con el mayor uso de anotaciones y atributos asociados.

Por otro lado señalaremos que inicialmente se plantea también la movilidad de POJOs para su uso en contextos distribuidos y desarrollo de aplicaciones cliente/servidor y/o web (recordad que están dentro de la especificación Java Enterprise Edition o JEE), aunque esta es una cuestión que no se abordará en esta asignatura.

La idea fundamental es que el uso de JPA no sea intrusivo, lo cual no quiere decir que sea transparente, pero exige poca modificación a las clases y en particular al cuerpo de los métodos de las clases, donde reside el código ejecutable.

Para la realización de consultas, JPA añade dos soluciones:

- Una primera basada en su propio lenguaje de consulta, aunque influenciada por SQL: Java Persistence Query Language (JP QL), heredando características de la API EJB QL, ya existente en la especificación EJB.
- Y como segunda opción una navegación en el modelo de objetos, sin utilizar los conceptos de columnas, claves primarias/foráneas y *joins*. Para ello utiliza una API particular denominada Criteria API.

Finalmente, JPA ofrece ciertas convenciones y directrices en el desarrollo, que facilitan su uso:

- Doble solución con la configuración XML vs. Anotaciones, a elección del programador.
- Gran peso a los valores por defecto: siguiendo el lema de **"convención sobre configuración"**.
- Facilidad para su integración en servidores de aplicaciones (JEE).
- Posibilidad de funcionamiento independiente de servidores e integración de pruebas automáticas.

3.3 Arquitectura

Con todos estos conceptos introducidos, podemos mostrar a continuación la arquitectura general con la que se trabajará en la asignatura (ver Ilustración 1):



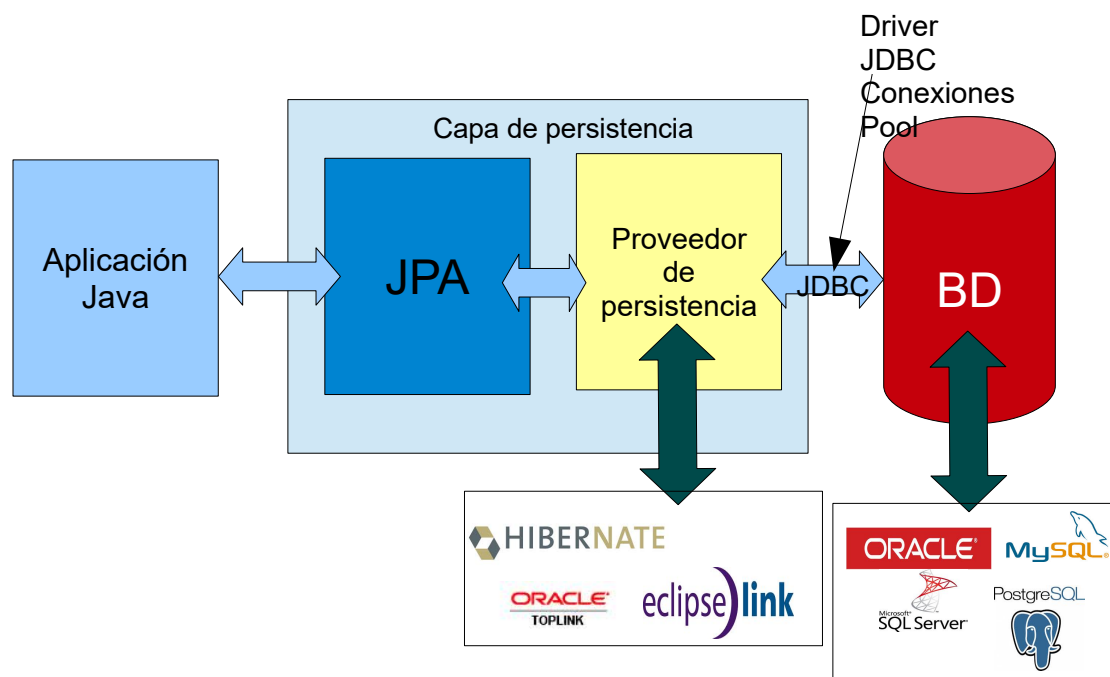


Ilustración 1: Arquitectura general con JPA en la asignatura

Como programadores, **nuestro trabajo** consistirá en construir el código correspondiente al bloque **Aplicación Java**. Este bloque está formado por el código del conjunto de entidades persistentes (mapping con la base de datos), clases de utilidad, clases de patrones de acceso a datos como DAO y el código que propiamente implementa la transacción o lógica de negocio.

En nuestro código utilizaremos solo elementos de **JPA** que permitirán enganchar con el *framework* de persistencia concreto, pero en la medida de lo posible, **sin atarnos a una solución concreta de implementación** (respecto al *framework* de persistencia).

La **Capa de persistencia** es implementada a través de **JPA** como interfaz (abstracta) – siguiendo la metáfora de enchufes en una instalación eléctrica real – y conectando en tiempo de ejecución con una implementación concreta de un **proveedor de persistencia** (e.g., Hibernate en nuestro caso en prácticas). A su vez al proveedor de persistencia se le indica declarativamente (en un fichero externo) el *driver* JDBC y los parámetros de la cadena de conexión a la BD.

Así pues, **si utilizamos correctamente JPA en nuestro código**, y no utilizamos código dependiente del proveedor (e.g. Hibernate) ni de la base de datos concreta (e.g. Oracle), el código es en mayor medida **independiente**, y podríamos posteriormente ejecutar nuestra aplicación con una nueva combinación de proveedor/SGBD (e.g., EclipseLink con MySQL).

Esto no siempre es fácil ni inmediato. Cada *framework* y SGBD incluyen ajustes, soluciones y optimizaciones propias, no siempre incluidos en la especificación JPA. Se produce la típica batalla entre "*soluciones muy generales pero no óptimas*", frente a la "*optimización con una solución concreta*" (atándonos a proveedores particulares, con los riesgos que tiene).

En esta asignatura, y en la parte correspondiente a JPA se intentará tomar, en la medida de lo posible, el primer enfoque, siendo conscientes de los pros y contras.

4. Entidades

Inicialmente abordaremos el problema de la persistencia entendiendo que una instancia u objeto (generado a partir de una clase) tiene una correspondencia 1 a 1 con una fila o registro de una tabla. Los



atributos se corresponden con columnas y viceversa. Las relaciones entre entidades introducen un nivel de complejidad superior que será abordado más adelante en otras secciones del tema.

Para convertir un POJO a entidad, se siguen las siguientes normas:

- La clase será anotada con `@Entity` (`javax.persistence.Entity`).
- Tendrá un constructor sin argumentos o constructor no-arg (`public` o `protected`).
- Sin atributos ni métodos persistentes con modificador `final`.
- Si se pasa como objeto remoto implementa `Serializable` (realmente se pasa por valor).
- Pueden heredar de entidades y no entidades.
- Los atributos persistentes NO pueden ser públicos: siempre se accede con *getters*, *setters* y métodos de negocio.

A continuación se muestra un ejemplo de cómo convertir una clase `Employee` en una entidad persistente, añadiendo un par de anotaciones (ver Código 1):

```
package examples.model;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Employee {

    @Id // Primary Key (PK)
    private int id;
    private String name;
    private long salary;

    public Employee() {}
    public Employee(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

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

    // more setters and getters using name, salary, etc.
    @Override
    public String toString() {
        return "Employee id: " + getId() + " name: " + getName() + " salary: " + getSalary();
    }

    // other methods...
}
```

Código 1: Ejemplo básico de entidad JPA

Como se puede comprobar en el ejemplo, la clase no es "*muy diferente*" a una clase tradicional en Java: atributos, métodos `get`, `set` y algún método típico como `toString()`. Si queremos que los objetos de `Employee` se correspondan con filas en la tabla `EMPLOYEE`, hay que modificar la clase con anotaciones (como podemos ver no es del todo transparente, pero el trabajo necesario tampoco es grande).



Simplemente añadiendo la anotación `@Entity` **justo delante de la declaración de la clase**, ya estamos indicando que la clase se transforma en una entidad persistente. Para finalizar el trabajo debemos indicar que el atributo `id` se corresponde con la clave primaria `ID` en la tabla `EMPLOYEE`. Sin más.

En este ejemplo se sigue el lema de "*convención sobre configuración*": se asume que el nombre de la clase/entidad y de la tabla coinciden, y que el nombre del atributo `id` y de la clave primaria en la tabla también coinciden. Dependiendo del grado de similitud de coincidencia entre nombres, nos tocará realizar más o menos trabajo.

Si no existiese esta coincidencia, hay que ampliar la información en la anotación, como se muestra en el siguiente ejemplo (ver Código 2). En dicho ejemplo se asume que la tabla y que los atributos tienen nombres diferentes o acortados (e.g. la tabla tendría la siguiente definición `EMP(EMP_ID (PK), NAME, SAL, COMM)`).

```
@Entity
@Table(name="EMP") // Example 1: table name EMP
public class Employee { ... }

@Entity
@Table(name="EMP", schema="HR") // Example 2: table and schema in Oracle
public class Employee { ... }

@Entity
@Table(name="EMP", catalog="HR") // Example 3: table and catalog in SQLServer
public class Employee { ... }

// Example 4 changing column names
@Entity
@Table(name="EMP")
public class Employee {
    @Id
    @Column(name="EMP_ID")
    private int id;
    private String name;
    @Column(name="SAL")
    private long salary;
    @Column(name="COMM")
    private String comments;
    // ...
}
```

Código 2: Ejemplo básico de entidad JPA sin correspondencia de nombres

JPA no dispone de anotaciones especiales para vistas, utilizando la misma anotación `@Table` (hasta la fecha no se ha incluido ninguna anotación especial para vistas). En estos casos seguimos limitados a las operaciones posibles sobre una vista, típicamente (aunque no exclusivamente) solo consultas, ignorándose las operaciones que modifiquen datos. Como curiosidad, el framework Hibernate añade una anotación `@Immutable`, que no es parte del estándar. En esta asignatura nos centraremos en el trabajo con tablas.

4.1 Acceso a atributos y propiedades

Llegado este punto puede surgirnos la duda: ¿cómo accede el *framework* a los valores de los atributos para crear la correspondiente SQL y mover los datos entre ambos "*mundos*"?

El acceso puede ser de dos formas (y no exclusivo):

1. Accediendo a los atributos:
 - A través de reflexión, en tiempo de ejecución.
 - Anotando el atributo que debe tener modificador distinto a `public`.



- Ej: `@Id private int id;`

2. Accediendo a propiedades:

- Con los correspondiente métodos `getX` y `setX` donde `X` se sustituye por el nombre del atributo (y métodos `isX` para consulta con booleanos).
- Los métodos deben ser `public` o `protected`.
- La anotación se coloca sobre el método `get`.
- Ej: `@Id public int getId() { return id; }`

Sin embargo en JPA se permite un acceso mixto, utilizando ambos mecanismos a la vez (atributo vs. propiedad/método). También se permite que las subclases redefinan el método de acceso (heredado) (e.g. `@Access(AccessType.FIELD)`).

Estos mecanismos no se aplican a datos temporales no persistentes (`transient` en Java) o propiedades marcadas con `@Transient`. En el siguiente ejemplo (ver Código 3), se muestra el uso combinado de estas soluciones.

```
@Entity
@Access(AccessType.FIELD)
public class Employee {
    public static final String LOCAL_AREA_CODE = "613";
    @Id private int id;
    @Transient private String phoneNum;
    ...

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getPhoneNumber() { return phoneNum; }
    public void setPhoneNumber(String num) { this.phoneNum = num; }

    @Access(AccessType.PROPERTY) @Column(name="PHONE")
    // exception of general case using field
    protected String getPhoneNumberForDb() {
        if (phoneNum.length() == 10)
            return phoneNum;
        else
            return LOCAL_AREA_CODE + phoneNum;
    }

    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3);
        else
            phoneNum = num;
    }
    ...
}
```

Código 3: Acceso básico combinado con atributos vs. propiedades

Inicialmente se ha indicado un acceso por atributo anotando la clase (`@Access(AccessType.FIELD)`). Dicho acceso se puede observar por ejemplo en el atributo `id`.

Sin embargo este comportamiento se cambia en el acceso al número de teléfono por parte de la base de datos (atributo `phoneNum`). En este caso, se accede al atributo a través de dos métodos: `getPhoneNumberForDb` (método anotado con `@Access`) y `setPhoneNumberForDb`. Además, al no existir coincidencia de nombre con los métodos, se ha indicado explícitamente la columna en la tabla – `@Column(name="PHONE")`. Esto permite “interceptar” los datos entre el modelo de objetos y la BD, realizando las operaciones de transformación sobre el número de teléfono. En este ejemplo concreto, cuando se quiere guardar el número de teléfono del objeto, en la BD, a través del método `get`, se añade o no el prefijo local si procede. Cuando se lee el valor de la BD a través del método `set`, se elimina el prefijo del valor a almacenar en el objeto, cuando procede. De esta forma ambos métodos **interceptan** y



transforman los valores. Cuando sean necesarias este tipo de transformaciones, es necesario utilizar el acceso con propiedades.

4.2 Tipos Java

Las restricciones en cuanto al uso de tipos Java en JPA se establece en:

- Tipos primitivos y clases de envoltura (*wrappers*) como `Long`, `Character`, etc.
- Cadenas de texto como `String`.
- Números largos como `java.math.BigInteger` y `java.math.BigDecimal`.
- Fechas como `java.util.Date` y `java.util.Calendar`
- Tipos de datos SQL para fecha y hora, como `java.sql.Date`, `java.sql.Time` y `java.sql.Timestamp`.
- Tipos serializables.
- Arrays de `byte[]`, `Byte[]`, `char[]` y `Character[]` (para tipos de datos `BLOB` o `CLOB`).
- Tipos enumerados Java.
- Clases embebidas propias de JPA (se verán posteriormente).

Para colecciones de datos se recomienda el uso de interfaces Java, y **preferiblemente con genericidad**:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

Nota: la especificación JPA 2.2 parece que incluirá ya algún método adicional que permita también la integración de la interfaz `java.util.Stream` incluida en la versión 8 de Java, pero actualmente no en JPA 2.1.

Las colecciones de tipos básicos de Java o embebidos, usan la anotación `@ElementCollection` con atributos de anotación `targetClass` (con tipo embebido se indica, pero se omite con tipos Java ya definidos) y `fetch` (por defecto `LAZY`, frente a `EAGER`). El atributo `fetch` se explicará en el siguiente apartado.

En el siguiente ejemplo (ver Código 4), se muestran distintos usos de los tipos de datos colección, remarcando el uso **no recomendado** de las soluciones sin genericidad (tipos *raw*) como el uso de `Collection` o `ArrayList`, sin dar un parámetro genérico actual:

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet<String>();
    ...

    @ElementCollection(targetClass=Phone, fetch=EAGER)
    protected Collection phones = new ArrayList(); // Not recommendable

    @ElementCollection(fetch=LAZY)
    protected List<Address> addresses = new ArrayList<Address>();
}
```

Código 4: Ejemplo de tipos de datos colección con JPA



Estas colecciones de elementos, se utilizan en relaciones uno a varios con valores cuya vida está ligada a la entidad fuerte de la que dependen, y cuyos datos están almacenados en otra tabla. En posteriores secciones del tema se profundizará sobre la relaciones más habituales entre entidades fuertes, más habituales (i.e., uno a uno, uno a varios o varios a varios).

4.3 Carga de valores

En JPA se permite indicar al *framework* si queremos que los valores se carguen de manera inmediata (ansiosa o `EAGER`) o por el contrario de manera diferida (perezosa o `LAZY`) al cargar los datos de una entidad.

La implementación de `LAZY` por parte del proveedor de persistencia es **opcional** (puede que lo haga o no) pero en el caso de `EAGER` es **obligatoria**.

Tiene sentido su uso si no es necesario traer todo el dato (o datos) inicialmente, y se puede esperar a su acceso real por parte de la aplicación para recuperar el valor de la base de datos (pudiera ser que nunca se accede al dato y por lo tanto supone un ahorro de recursos). El propio *framework* controla el acceso al atributo y la posible necesidad de acceder al valor.

Esto es particularmente útil con datos binarios o de texto grandes – donde se combina con la anotación `@Lob` – (tipos `BLOB`, `CLOB`, etc. en base de datos), ver Código 5, o con las filas relacionadas a otra entidad persistente.

```
@Entity
public class Employee {
    @Id
    private int id;
    @Basic(fetch=FetchType.LAZY)
    @Lob @Column(name="PIC")
    private byte[] picture;
    // ...
}
```

Código 5: Ejemplo de vinculación perezosa de un dato binario grande (BLOB)

Nota: la anotación `@Basic` se puede utilizar **opcionalmente** sobre cualquier propiedad persistente o variable de instancia de tipo primitivo, envoltorio (wrappers), `java.lang.String`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, enumerados, y cualquier otro tipo que implemente `Serializable` (ver especificación de JPA 2.1). Permite redefinir el valor por defecto de `fetch` (con valor por defecto `Fetch.EAGER`) y también especificar el valor booleano de la pista (*hint*) `optional` para indicar si el valor podría valer `null` o no (ignorado su uso para tipos primitivos). Dado su carácter opcional, solo se utilizará en aquellos casos en los que sea necesario realmente redefinir el valor por defecto, normalmente del `fetch`.

4.4 Clases embebidas

Se entiende por una clase embebida a aquellas **entidades especiales que no tienen identidad, ligadas a la entidad fuerte que las contiene**, permitiendo su uso recursivamente (una clase embebida puede estar compuesta de otras). Por otro lado pueden establecer relaciones con otras entidades y colecciones.

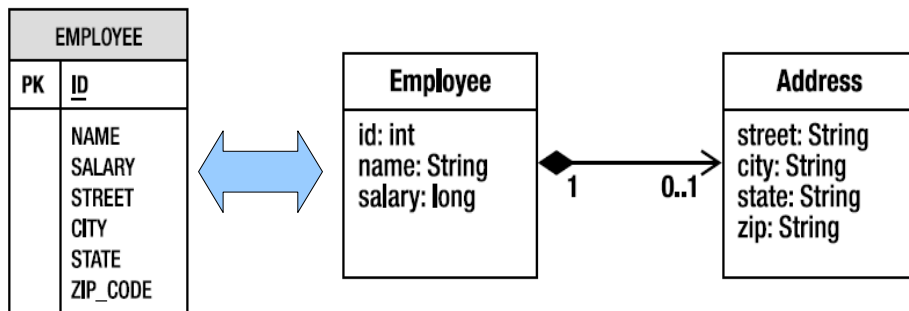
La anotación utilizada en la **declaración** de estas clases es `@Embeddable` en lugar de `@Entity`.

Para su **uso** (no requerido) en otras clases se utiliza `@Embedded`.

Tomemos como ejemplo el Dibujo 1, donde una tabla `EMPLOYEE` contiene campos que definen su dirección. Esto se transforma en el diagrama de clases en una composición entre `Employee` y `Address`



(rombo negro) con navegabilidad de `Employee` hacia `Address` (punta de flecha), pero no al revés. La multiplicidad 0..1 indica que no se registra dirección para todo empleado, pudiendo estar su valores vacíos o nulos.



Dibujo 1: Ejemplo de clase embebida. Figura extraída de [Keith & Schincariol, 2013]

La dirección no tiene como tal un identificador o clave primaria, su identificación (y su vida) está ligada al empleado correspondiente, y por lo tanto su rol en JPA es el de una clase (tipo) embebido.

Para representar esta situación en JPA se utilizaría el siguiente código parcial (ver Código 6):

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}

@Embeddable @Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
    // ...
}
```

Código 6: Uso de clases (tipos) embebidos

4.5 Claves primarias: simples vs. compuestas

Un concepto fundamental en las bases de datos, es el de **clave primaria** en nuestras entidades. La correspondencia de dicho concepto en JPA, como ya se ha visto, es a través de la anotación `@Id`.

Los tipos que se pueden utilizar para la clave primaria son tipos primitivos, *wrappers*, `java.lang.String`, `java.util.Date`, `java.sql.Date`, `java.math.BigDecimal` y `java.math.BigInteger`.

Aunque se pueden utilizar, se desaconseja el uso `float`, `double`, `Float`, `Double` y `BigDecimal`, dado que debido a errores de redondeo, pueden generar problemas a la hora de realizar comparaciones con el método `equals`².

Sin embargo, la clave primaria no siempre se compone de un único campo/atributo (clave simple), dándose la posibilidad de que sea compuesta por varios.

Para construir las **claves compuestas** se dan dos variantes o posibilidades:

1. Clase identidad: `@IdClass`
2. Identificador embebido: `@EmbeddedId`

En ambas soluciones, las clases que componen la "identidad" y que se usan como clave primaria tienen unas reglas a cumplir específicamente:

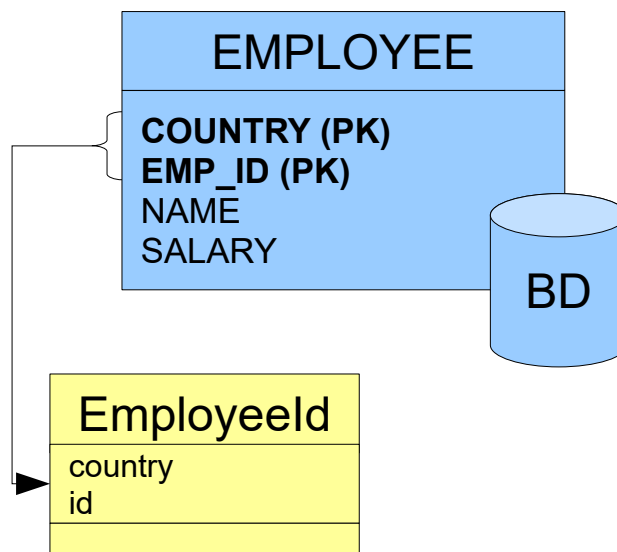
- Son clases públicas, con propiedades públicas o protegidas.

2 Recordad que el método `equals` se hereda de `Object`, se suele redefinir y permite comparar el estado de los objetos, no sus identidades. Es el método utilizado por JPA para comparar objetos entre sí.



- Tienen un constructor sin argumentos público.
- Redefinen `hashCode` y `equals`.
- Implementan la interfaz `Serializable`.
- Vinculadas a:
 - Atributos de la entidad (coinciden en nombre) en la `@IdClass`.
 - Clase embebida (`@EmbeddedId`).

A continuación, partiendo de la situación mostrada en Dibujo 2, donde se muestra la tabla `EMPLOYEE` con una clave compuesta formada por `COUNTRY` y `EMP_ID`, se resuelve su implementación con ambas soluciones a continuación.



Dibujo 2: Tabla con clave primaria compuesta y clase identidad asociada

En primer lugar, utilizando **clases identidad**. En el Código 7, se muestra la clase/entidad `Employee`. Se indica que tiene una clave compuesta implementada en una clase identidad, a través de la anotación de clase `@IdClass`, con valor adicional la clase que implementa dicha clave compuesta (`EmployeeId.class`).

```

@Entity
@IdClass(EmployeeId.class)
public class Employee {

    @Id private String country;
    @Id @Column(name="EMP_ID")
    private int id;

    private String name;
    private long salary;
    // ...
}
  
```

Código 7: Clase con clave primaria compuesta implementada en clase identidad

La clase que implementa la clave compuesta (`EmployeeId`) (ver Código 8), debe cumplir las reglas generales ya expuestas, y además **debe existir** una correspondencia uno a uno entre sus atributos y los atributos marcados en la entidad persistente con `@Id`.



En nuestro ejemplo concreto los atributos, `country` e `id` en `Employee` tienen sus correspondientes réplicas en la clase `EmployeeId`, uno a uno. Como curiosidad señalar la ausencia de anotaciones JPA en la clase identidad.

```
public class EmployeeId implements Serializable {
    private String country;
    private int id;
    public EmployeeId() {} // no-args constructor, mandatory
    public EmployeeId(String country, int id) {
        this.country = country;
        this.id = id;
    }
    // without setters... IMMUTABLE
    public String getCountry() { return country; }
    public int getId() { return id; }

    // utility methods for identity... (inherited and overridden from Object)
    @Override
    public boolean equals(Object o) {
        return ((o instanceof EmployeeId) &&
            country.equals(((EmployeeId)o).getCountry()) &&
            id == ((EmployeeId)o).getId());
    }
    @Override
    public int hashCode() {
        return country.hashCode() + id;
    }
}
```

Código 8: Clase identidad para clave primaria compuesta

Existe una segunda solución, usando **identificadores embebidos**. En esta solución (ver Código 9) se utiliza un atributo (**y sólo uno**) en la clase `Employee` de tipo embebido, indicándose su papel de clave primaria con la anotación `@EmbeddedId`.

A continuación se implementa la clase embebida siguiendo las reglas generales para las claves compuestas. En este caso la **clase embebida** sí contiene anotaciones JPA.

```
@Entity
public class Employee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;
    // ...
}

@Embeddable
public class EmployeeId implements
Serializable {
    private String country;

    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}

    public EmployeeId(String country, int id) {
        this.country = country;
        this.id = id;
    }

    // Getter methods, equals() and hashCode()
    // implementations, are the same as @IdClass
    // example
}
```

Código 9: Clase con clave primaria compuesta implementada con clase embebida



4.6 Claves primarias: generación de valores únicos

Un problema habitual con las claves primarias es la generación de valores únicos. En un SGBD se suele incorporar algún mecanismo que facilita su generación, y por lo tanto, es necesario estudiar cómo se integran las distintas soluciones dentro de JPA.

La solución adoptada es indicar en la clave primaria, la **estrategia** de generación utilizada por el SGBD. Se proporcionan cuatro:

1. AUTO: el proveedor de persistencia proporciona el id. Sin importar la estrategia. Suele necesitar permisos DBA (para crear recursos). **Nota: se usa en producción y prototipos pero no en explotación. No utilizar.**
2. TABLE: utiliza una tabla con dos columnas, 1) nombre del "generador de secuencia" y 2) valor (último id). Cada generador es una fila de la tabla, simulando el uso de secuencias. Se puede especificar el nombre de la tabla, del generador, columna clave y valor. Además se pueden dar parámetros adicionales como el valor inicial (`InitialValue`, valor por defecto 0) y una caché de valores (`AllocationSize`, valor por defecto 50). Es una solución portable, pero no recomendable si nuestra BD soporta secuencias o autonuméricos.
3. SEQUENCE: mecanismo interno de la BD (e.g., Oracle, PostgreSQL) para generar identificadores. Tiene un alto rendimiento.
4. IDENTITY: mecanismo interno de la BD generalmente a través de valores autonuméricos (e.g., MySQL o SQLServer). Menor rendimiento que las secuencias.

A continuación se presentan los correspondientes códigos de ejemplo para los cuatro casos. En el primer caso (ver Código 10) se delega en el proveedor de persistencia el mecanismo de generación de valores. Aunque es muy simple se recomienda no utilizarlo.

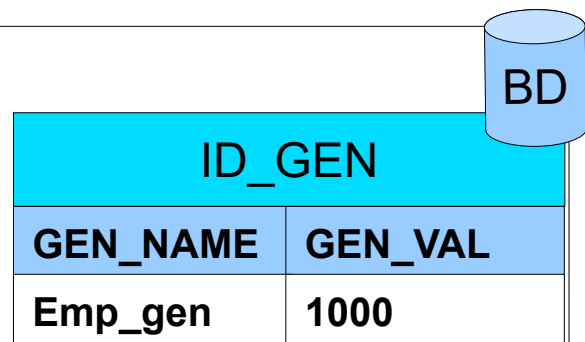
```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    // ...
}
```

Código 10: Generación con AUTO

En el segundo caso (ver Código 11), se utiliza una tabla auxiliar para generar los identificadores. Es necesario indicar el nombre de la tabla (e.g. `ID_GEN`) y nombre del generador (e.g. `Emp_Gen`) así como los nombres de las columnas que contienen el nombre del generador y el valor actual (e.g. `GEN_NAME` y `GEN_VAL` respectivamente). Es una solución muy portable, puesto que no depende de mecanismos particulares del SGBD.

```
// default values taken by provider...
@Id @GeneratedValue(strategy=GenerationType.TABLE)
private int id;

// all values...
@TableGenerator(name="Emp_Gen", table="ID_GEN",
    pkColumnName="GEN_NAME",
    valueColumnName="GEN_VAL",
    initialValue=1000,
    allocationSize=100)
@Id @GeneratedValue(generator="Emp_Gen")
private int id;
```



ID_GEN	
GEN_NAME	GEN_VAL
Emp_gen	1000

Código 11: Generación con TABLE



```
// default values taken by provider...
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE)
private int id;

// settings with sequence name
@SequenceGenerator(name="Emp_Gen",sequenceName="Emp_Seq")
@Id @GeneratedValue(generator="Emp_Gen")
private int getId;
```

Código 12: Generación con SEQUENCE

La tercera opción es utilizar el concepto de secuencias que existe en algún SGBD. Genera automáticamente nuevos identificadores, resolviendo conflictos propios del uso concurrente. Ejemplo típico es Oracle, utilizado en esta asignatura.

Finalmente, la última opción, usando una estrategia de identidad, generalmente resuelta por el SGBD a partir de la generación de valores auto-numéricos, como por ejemplo en MySQL o SQLServer.

```
// default values without generator
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

Código 13: Generación con IDENTITY

En general, el uso de un SGBD concreto suele influir mucho (quizás demasiado) en la utilización de una u otra estrategia. Dado que en la asignatura los ejemplos y ejercicios se resuelven con Oracle, se utilizará preferiblemente la solución basada en secuencias, por simplicidad y rendimiento. Aunque no sea la solución más portable.

4.7 Tipos enumerados

Tomamos una definición clásica de tipo enumerado como "*tipo de datos con un conjunto de valores acotado y que implícitamente tienen también asignado un valor ordinal*". Este valor ordinal puede ser utilizado el almacenamiento y acceso al valor.

Ejemplos típicos son los días de la semana (e.g., Lunes, martes, miércoles, etc.) o los palos de la baraja (e.g., Oros, Copas, Espadas y Bastos).

En Java, desde su versión 1.5 se incluyó este concepto de tipo enumerado y por lo tanto se utilizará para almacenar valores que se ajusten a la definición previa en JPA (ver Código 14).

Habitualmente se utiliza un valor almacenado de tipo entero en la BD, para almacenar el valor correspondiente del tipo enumerado (solución basada en posición). Esta solución es óptima en espacio, pero origina una serie de problemas si el tipo enumerado cambia o se reordena.

```
public enum EmployeeType {
    FULL_TIME_EMPLOYEE,    // ordinal 0
    PART_TIME_EMPLOYEE,    // ordinal 1
    CONTRACT_EMPLOYEE      // ordinal 2
}

@Entity
public class Employee {
    @Id private int id;
    private EmployeeType type; // valor por defecto @Enumerated(EnumType.ORDINAL)
    // ...
}
```

Código 14: Ejemplo de tipo enumerado con ordinal y uso en atributo de entidad



Como alternativa al uso del ordinal, se plantea el uso del texto correspondiente en la BD. Esta solución no sufre de los problemas por cambio y reordenación, pero es menos eficiente y sigue siendo vulnerable a cambio en los textos descriptivos. Para llevar a cabo esta solución se utilizan las anotaciones `@Enumerated` y valor enumerado `EnumType` (ver Código 15).

```
// Using previous definition of EmployeeType

@Entity
public class Employee {
    @Id
    private int id;
    @Enumerated(EnumType.STRING)
    private EmployeeType type;
    // ...
}
```

Código 15: Ejemplo de tipo enumerado con texto y uso en atributo de entidad

Si se no se utiliza la anotación `@Enumerated` o no se especifica el valor `EnumType`, se supone el valor por defecto, que es `EnumType.ORDINAL` con valor entero.

Se recuerda además que aunque los `enum` en Java pueden incluir atributos y métodos adicionales (son básicamente clases Java), **JPA no da soporte a estos elementos adicionales.**

4.8 Tipos temporales

A la hora de trabajar con valores de tipo fecha y hora, se establecen mapeos especiales entre atributos y campos. El mapeo adicional se produce en los campos genéricos de tipo `java.util.Date` y `java.util.Calendar`, **no siendo** necesario en los tipos de datos específicos `java.sql.Date`, `java.sql.Time` ni `java.sql.Timestamp`.

Para los dos tipos de datos genéricos, en frameworks como Hibernate, se toma valor por defecto *timestamp* con nanosegundos. Para cambiar dicho valor se deben añadir la anotación `@Temporal`. Junto con la anotación, se debe indicar el valor correspondiente de mapeo SQL según se quiera:

- Almacenar solo la fecha (día, mes y año) con `TemporalType.DATE`.
- Almacenar solo el tiempo (sin nanosegundos) con `TemporalType.TIME`.
- Almacenar fecha y hora (con nanosegundos) con `TemporalType.TIMESTAMP`.

En el Código 16 se muestran algunos ejemplos de uso con `TemporalType.DATE` (su uso con los otros valores es similar sintácticamente, se omite por brevedad).

```
@Entity
public class Employee {
    @Id
    private int id;
    @Temporal(TemporalType.DATE)
    private java.util.Calendar dob;
    @Temporal(TemporalType.DATE)
    @Column(name="S_DATE")
    private java.util.Date startDate;
    // ...
}
```

Código 16: Ejemplo de mapeo especial para tipo de dato temporal

En la especificación 2.2 de JPA está anunciado el soporte de los tipos del paquete `java.time` incluidos en la versión Java 8, como parte de su "Date and Time API". Algún framework como Hibernate da soluciones



particulares desde su versión 5, pero no son objeto de estudio ni uso en la asignatura, dado que no son parte del estándar ni portables.

4.9 Datos transitorios

Nos podemos encontrar con atributos que existen en el modelo de objetos **pero que no tienen un reflejo directo (ni deben) en la base de datos**. Estos datos solo están en memoria y se pierden finalmente al eliminar el objeto.

En Java se indica esto, utilizando un modificador `transient`, que es una palabra reservada del lenguaje (ojo, tampoco se serializan estos atributos).

En JPA, se proporciona un mecanismo equivalente adicional con la anotación `@Transient`, evitando que se realicen mapeos sobre estos atributos en la base de datos. Puede ser útil con datos calculados o derivados, que no tienen un reflejo directo en la base de datos.

4.10 Anotaciones de columna

Como se ha mostrado en algún ejemplo previo, existe una anotación adicional para el mapeo de atributos denominada `@Column`.

Esta anotación da **información adicional** relativo al mapeo físico con la base de datos y es particularmente útil si se quiere generar el DDL de las tablas (crearlas) a partir de la información recogida en el modelo de clases, pero no cumplen un papel de validadores. Dichas comprobaciones, o bien se realizan a nivel de la base de datos (mediante las *constraints*) o bien mediante el uso adicional de APIs de validación. **Esto es meramente informativo para la generación del DDL y útil para ingeniería inversa desde el modelo de clases hacia la base de datos** (pero en esta asignatura siempre trabajaremos desde la base de datos hacia el modelo de clases).

Por ejemplo se permite la inclusión de información adicional sobre valores nulos permitidos (`nullable`), unicidad (`unique`), longitud para cadenas (`length`) o decimales en numéricos (`scale`).

Sin embargo, su uso sí es obligado cuando los **nombres de los atributos no coinciden con los nombres de los campos correspondientes en la tabla**, de manera similar al uso de `@Table`. Por ejemplo:

```
@Entity
@Table("EMP") // suponemos que la tabla se llama distinto que la entidad
public class Employee {
    @Id
    @Column(name = "IDEN") // suponemos que el campos se llama distinto que el atributo
    private int id;
    @Column(name = "NAME", nullable = false, length = 150) // información adicional sobre el campo
    private String nombre;
    // ...
}
```

Código 17: Ejemplo de mapeo adicional con `@Column`

Existen dos atributos adicionales para indicar la posible inserción (`insertable`) o actualización (`updatable`) de la columna (valores booleanos `true/false`) por parte del proveedor. Esto permite configurar, en cierta manera, valores de solo lectura. Pero esto no impide que los valores no puedan ser modificados temporalmente en memoria. Así pues, se aplica esto solo en el momento de traducir los cambios al correspondiente SQL enviado a la base de datos, sin lanzar ninguna excepción de aviso (aunque esto puede ser dependiente del proveedor).

Dado que en esta asignatura, se partirá generalmente de una base de datos ya definida, con el DDL correspondiente ya creado, solo se insistirá en esta anotación `@Column` cuando sea estrictamente



necesario. En este mismo sentido, no se profundizará en anotaciones de mapeo similares, como `@UniqueConstraint`, `@Index` o `@ForeignKey` quedando fuera del objetivo de esta asignatura.

5. Resumen

En esta sección se han planteado el concepto, propiedades y arquitectura básica de JPA. Por otro lado se han planteado las reglas básicas de mapeo de entidades JPA, entre el modelo de objetos y la base de datos, con el mapeo básico de atributos, teniendo en cuenta los tipos Java, las políticas de carga de valores (ansiosa vs. perezosa) y el uso de clases embebidas.

Se ha abordado el uso de claves primarias simples y compuestas, la generación de identificadores únicos para dichas claves, y algún uso más avanzado como el de tipos enumerados, tipos temporales, datos transitorios y anotaciones de columna.

Sin embargo, en esta sección se ha simplificado el problema, afrontando el problema del mapeo entidad-tabla, uno a uno. En la práctica sabemos que las entidades están relacionadas entre sí y no son algo aislado. En la siguiente sección se describen las soluciones de mapeo para la relaciones.

6. Glosario

API: Application Programming Interface o interfaz de programación de aplicaciones.

JPA: Java Persistence API

POJO: Plain Old Java Object u objeto Java plano.

SGBD: Sistema Gestor de Bases de Datos

7. Bibliografía

[Keith & Schincariol, 2013] Mike Keith & Merrick Schincariol. ***Pro JPA 2. A definitive guide to mastering the Java Persistence API (2013)*** Apress. 2nd edition.

[Keith et al., 2018] Mike Keith, Merrick Schincariol, Massimo Nardone. ***Pro JPA 2 in Java EE 8. An In-Depth Guide to Java Persistence APIs.*** (2018) Apress. 3rd edition.

[Oracle, 2017] The Java EE 8 Tutorial (2017). Part VIII. Persistence. Disponible en <https://javaee.github.io/tutorial/toc.html>

[Oracle, 2014] The Java EE 7 Tutorial (2014). Part VIII. Persistence. Disponible en <http://docs.oracle.com/javaee/7/tutorial/>

8. Recursos

Especificaciones públicas:

[Oracle, 2013] JSR 338: JavaTM Persistence API, Version 2.1 (2013). Version: Final Release http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf



Bibliografía complementaria:

[Oracle, 2013] Oracle (2013) Toplink JPA Annotation Reference. Disponible en

<http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-096251.html>

Java Persistence. (2015, August 17). Wikibooks, The Free Textbook Project. Retrieved 12:00, March 5, 2017 from https://en.wikibooks.org/w/index.php?title=Java_Persistence&oldid=2984965.



Licencia

Autores: Jesús Maudes & Raúl Marticorena & Mario Martínez

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Civil

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2020



Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en <http://creativecommons.org/licenses/by-nc-sa/4.0/>

