Trabajando con JPA

Los identificadores de las entidades

Índice del capítulo

Introducción

- Como ya hemos comentado, las **entidades** deben tener un **atributo identificador** que las identifique de forma **unívoca**.
- Se trata de un concepto equivalente al de **clave primaria** de las tablas en SQL. Pese a que existen bases de datos relacionales que permiten la definición de tablas sin claves primarias, nosotros las vamos a **necesitar**, o bien una columna que cumpla con los mismos requisitos: **única y no nula**.
- La **elección y configuración** de identificadores **no es complicada** y está condicionada por la base de datos que usemos.

Elección de identificador

- Es posible que en una **entidad** tengamos **varios atributos que puedan utilizarse** como **identificador**, ya sea de forma conjunta (las claves múltiples que veremos más adelante) o independiente.

- En el mundo relacional, estas opciones se denominan **claves candidatas**. Por ejemplo, en el supuesto de una persona, podría ser su **dirección de correo electrónico**, **número de la seguridad social**, de móvil, etcétera.

- Sin embargo, la mejor forma de **seleccionar un identificador es no hacerlo**, sino **crear un atributo** específico para este menester consistente en **un número sin ningún significado en particular**.

Elección de identificador

- Estas son las ventajas:
  - Cualquier **registro** de cualquier tabla se identifica con un **número único** para esa tabla, o incluso toda la base de datos. Nos da igual **el diseño de las entidades** porque trabajaremos con ellas del mismo modo: identificándolas por un número.
  - A veces, un **valor considerado como único** no lo es o deja de serlo.
  - El identificador lo puede generar de forma **automática**, **eficiente** y **segura** la base de datos. De hecho, es la costumbre habitual.
  - Aunque se **modifiquen** los datos de un registro, siempre **se identificará por el mismo número**. Si en la tabla usuarios elegimos, por ejemplo, como clave primaria el atributo email, este valor es susceptible de cambiar. Pero si el usuario tiene el id 27, siempre será el usuario 27. Las bases de datos suelen permitir la modificación de las claves primarias, pero no es recomendable.
  - **Evitamos** la molestia de tener **claves primarias múltiples** formadas por más de una columna. Téngase en cuenta que las claves primarias son referenciadas por las ajenas, lo que nos complicará además la declaración de estas últimas.

Elección de identificador

- A este tipo de claves se les denomina **subrogadas** o, con menos frecuencia, **artificiales**.
- Por el contrario, las que **sí lo tienen**, son conocidas como **claves naturales o de negocio**.
- Si no estamos trabajando con una **base de datos preexistente** a la que nos tenemos que adaptar, se recomienda optar siempre por las **subrogadas** debido a sus **beneficios**.

Los identificadores

- Los **identificadores** en Hibernate representan la **clave principal de una entidad**.
- Esto implica que los valores son únicos para que puedan identificar una entidad específica, que **no sean nulos** y que **no se modificarán**.
- Hibernate proporciona formas diferentes de definir identificadores.

Los identificadores

- La forma más sencilla de definir un identificador es mediante la **anotación @Id**.
- Los **identificadores simples** se asignan mediante @Id a una sola propiedad. Se admiten los siguientes tipos:
  - Los tipos primitivos byte, int, short, long, float, double y char
  - Los wrappers en clases de los anteriores: Byte, Integer, Short, Long, FLoat, Double y Character.
  - BigInteger y BigDecimal.
  - Temporales de tipo Date y java.util.Date.
  - Cadenas.

Los identificadores

- Veamos un ejemplo rápido de definición de una entidad con una clave principal de tipo long:

```
@Entity
public class Student {

   @Id
   private long studentId;

   // standard constructor, getters, setters
}
```

Los identificadores

- Como cualquier atributo persistible, **no puede ser final** y se personaliza con la **anotación @Column** (@Id no tiene atributos).
- Puede heredarse, pero para ello hay que configurar cómo debe tratarse la jerarquía.
- No es necesario definir la columna como única y no nula, pues estas restricciones ya están implícitas.

Generando identificadores de forma automática

- Introducción.
- AUTO.
- IDENTITY.
- SEQUENCE.
- TABLE.

Introducción

- Si queremos generar **automáticamente** el valor de la clave principal, podemos añadir la **anotación @GeneratedValue**.
- Esto puede usar **cuatro tipos** de generación:
  - AUTO,
  - IDENTITY,
  - SEQUENCE y
  - TABLE.
- Si no especificamos explícitamente un valor, el tipo de generación **por defecto es AUTO**.

IDENTITY

- Este tipo de generación se basa en **IdentityGenerator**, que espera valores generados por una columna de identidad en la base de datos. Esto significa que se incrementan automáticamente.
- Para usar este tipo de generación, solo necesitamos configurar el **parámetro strategy**:

```
@Entity
public class Student {

   @Id
   @GeneratedValue (strategy = GenerationType.IDENTITY)
   private long studentId;
}
```

IDENTITY

- En **versiones antiguas de Oracle** no contamos con el tipo identidad.
  - MySQL\MariaDB:AUTO_INCREMENT
  - PostgreSQL: serial (es un tipo identidad simulado con una secuencia)
  - Oracle pre 12c: No disponible.
  - Oracle 12c: IDENTITY.
  - SQL Server: IDENTITY.

IDENTITY

- El **inconveniente** de esta estrategia es que **no se conoce el identificador de un nuevo registro** hasta que se ejecute la sentencia **INSERT**.

- Esto supone un **problema de eficiencia** porque Hibernate tiene que insertar inmediatamente el registro para poder asignar el identificador a la entidad, lo que impide la realización de ciertas optimizaciones y las **inserciones masivas en lotes** (batch).

SEQUENCE

- Una **secuencia** es un **objeto o tipo de datos** (la nomenclatura depende de la base de datos) que proporciona **valores numéricos únicos**.
- Almacena un valor que **aumenta según una cantidad de incremento** cuando se solicita un nuevo valor.
- Esto último se realiza con una **consulta SELECT**.

```
select nextval('hibernate_sequence')
```

SEQUENCE

- Con el siguiente código, indicamos a Hibernate que debe obtener el valor del identificador de las nuevas entidades empleando una secuencia.

```
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE)
private Long id;
...
```

- Si no personalizamos la secuencia, **Hibernate 5**, crea una única secuencia por esquema denominada **hibernate_sequence.** Esto implica que los valores generados serán únicos **para todas las entidades** en las que establezcamos su identificador del modo anterior.

- En Hibernate 6, este comportamiento cambió y se crea una secuencia para cada entidad, nombrada con el **nombre de la entidad y el sufijo _SEQ** (por ejemplo, expense_SEQ). Se puede volver al funcionamiento anterior activando el modo legacy en la configuración del fichero persistence.xml.

```
<property name="hibernate.id.db_structure_naming_strategy" value="legacy" />
```

SEQUENCE

- Ejemplo básico:

```
@Entity
public class Expense {

  @Id
  @SequenceGenerator(name="expense", sequenceName="expense_seq")
  @GeneratedValue(strategy=GenerationType.SEQUENCE,  generator="expense")
  private Long id;
```

SEQUENCE

- Para usar una identificación basada en secuencias, Hibernate proporciona la clase **SequenceStyleGenerator**.
- Este generador **utiliza secuencias si nuestra base de datos las admite**. Cambia a la generación de tablas si no son compatibles.
- Para personalizar el nombre de la secuencia, podemos usar la **anotación @GenericGenerator** con la estrategia SequenceStyleGenerator.

SEQUENCE

- Ejemplo:

```
@Entity
public class User {
   @Id
   @GeneratedValue(generator = "sequence-generator")
   @GenericGenerator(
    name = "sequence-generator", strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
    parameters = {
     @Parameter(name = "sequence_name", value = "user_sequence"),
     @Parameter(name = "initial_value", value = "4"),
     @Parameter(name = "increment_size", value = "1")
     }
   )
   private long userId;
}
```

SEQUENCE

- Ejemplo:
  - En este ejemplo se establece un **valor inicial para la secuencia**, lo que significa que la generación de la clave principal **comenzará en 4**.
  - **SEQUENCE** es el **tipo de generación recomendado** por la documentación de Hibernate.
- Los **valores generados son únicos** por secuencia. Si no especificamos un **nombre de secuencia**, Hibernate reutilizará la misma **hibernate_sequence para diferentes tipos**.

SEQUENCE

- Las bases de datos más populares tienen secuencias…con una llamativa excepción.
- **MySQL: No disponible.**
- **MariaDB pre 10.3: No disponible.**
- MariaDB 10.3: Sí.
- PostgreSQL: Sí.
- Oracle: Sí.
- **SQL Server pre 2012: No disponible.**
- SQL Server 20102: Sí.

SEQUENCE

- Vemos que en **MySQL** no hay secuencias.
- Cuando indicamos este tipo de identificador, Hibernate (*) simulará cada secuencia con una tabla con una única columna (next_val) que contendrá el último valor entregado para la secuencia simulada. Este «apaño» es una mala idea, y en breve explicaré el motivo cuando hable del generador tabla.
- Para ser rigurosos, debemos dejar claro que en **Hibernate 4** simplemente **se lanza una excepción** de tipo MappingException que abortará el inicio de la aplicación.
- Los **identificadores basados en secuencias** son los más **recomendables** porque permiten a Hibernate **averiguar el valor para una nueva entidad antes de su inserción**, tan solo tiene que pedir a la base de datos el siguiente valor de la secuencia. Esto posibilita **retrasar el momento de ejecución de los INSERT para mejorar el rendimiento** y también realizar **inserciones por lotes**.

TABLE

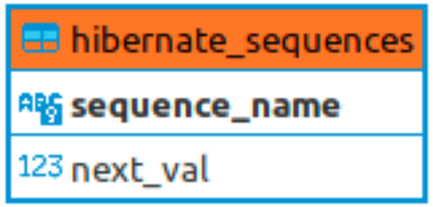
- Esta estrategia es **portable** a cualquier base de datos porque emplea SQL estándar.
- Utiliza **una tabla con dos columnas**, una de tipo **texto** con el nombre que identifica al «generador» de claves primarias, y otra **de tipo numérico** con el último valor que fue proporcionado por ese generador.
- Cuando **necesite una nueva clave primaria** del generador, Hibernate **la obtiene sumando uno** a ese valor.
- En la práctica, es lo que parece: **se están simulando secuencias**.

TABLE

- Ejemplo:

```
@Entity
@Table(name="cities")
public class City {
   @Id
   @GeneratedValue(strategy = GenerationType.TABLE)
   private Long id;
```

- En City se ha indicado que queremos usar **el modo tabla sin proporcionar ninguna configuración** adicional, como por ejemplo, **el nombre del generador**. Hibernate espera la existencia de la siguiente tabla con un único registro para calcular las claves primarias de aquellas entidades que usen el generador por defecto, denominado **default**.



	sequence_name	next_val
1	default	0

TABLE

- TableGenerator utiliza una **tabla de base de datos** subyacente que contiene segmentos de valores de generación de identificadores. Podemos personalizar el nombre de la tabla usando **la anotación @TableGenerator**:

```
@Entity
public class Department {
   @Id
   @GeneratedValue(strategy = GenerationType.TABLE, generator = "table-generator")
   @TableGenerator(name = "table-generator", table = "dep_ids", pkColumnName = "seq_id", valueColumnName = "seq_value")
   private long depId;
}
```

- En este ejemplo, podemos ver que también podemos **personalizar otros atributos** como **pkColumnName** y **valueColumnName**.

- Sin embargo, la desventaja de este método es que **no escala bien** y puede afectar **negativamente** al rendimiento.

TABLE

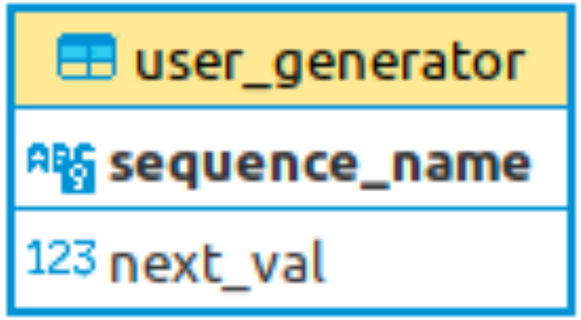
- Veamos un ejemplo:

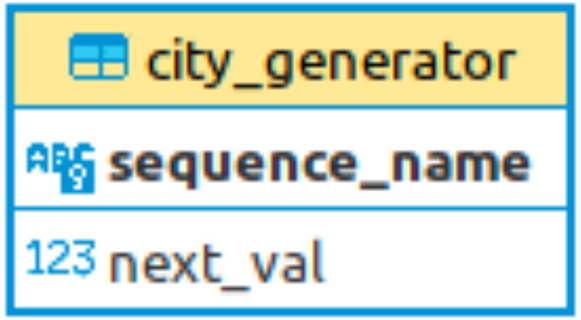
```
@Entity
@Table(name="cities")
public class City {
   @Id
   @TableGenerator(name="city_generator")
   @GeneratedValue(strategy = GenerationType.TABLE, generator = "city_generator")
   private Long id;
```

```
@Entity
@Table(name="users")
public class User {
   @Id
   @TableGenerator(name="user_generator")
   @GeneratedValue(strategy = GenerationType.TABLE, generator = "user_generator")
   private Long id;
```

TABLE

- Tendremos estas tablas:





TABLE

- La **propiedad table** puede ser muy útil. Nos permite definir el nombre de la tabla que contiene el generador.
- Permite **usar siempre la misma tabla** y que **cada generador sea un registro de la misma**. La idea es centralizar todos los generadores y evitar tener muchas tablas de este tipo (user_generator, city_generator…).

```
@Id
@TableGenerator(name="user_generator", table = "table_generator")
@GeneratedValue(strategy = GenerationType.TABLE, generator = "user_generator")
private Long id;
```

table_generator | Enter a SQL expression to

Grid

	sequence_name	next_val
1	city_generator	2240
2	user_generator	144

TABLE

- Otras propiedades son:
  - **initialValue**. El número en el que comienzan los identificadores. Por omisión es cero para que el primer valor sea uno.
  - **allocationSize**. El incremento de la «secuencia» después de asignarse un nuevo valor. Funciona igual que el allocationSize de @SequenceGenerator, así que no repetiré mis observaciones.
  - **pkColumnName**. Nombre de la columna que almacena el nombre del generador. Por omisión, Hibernate usa sequence_name.
  - **pkColumnValue**. El nombre del generador dentro de la columna pkColumnName. Hibernate usa el nombre de la tabla de la entidad.
  - **valueColumnName**. El nombre de la columna con la última clave asignada para el generador. Hibernate usa next_val.

TABLE

- La **flexibilidad** que aporta esta estrategia tiene un **precio alto**, hasta tal punto de no ser recomendable: resulta **ineficiente** porque hay que **consultar y actualizar** una tabla para obtener la clave, además de gestionar los posibles problemas de concurrencia de estas operaciones.
- Esto da lugar a un **pequeño cuello de botella** a la hora de insertar los registros.
- Si buscamos la máxima compatibilidad de nuestros identificadores con diversas bases de datos, es aconsejable recurrir a la estrategia que estudiaremos a continuación, aunque tampoco está exenta de inconvenientes.

AUTO

- Es la **estrategia por defecto** y, al igual que la anterior, **portable**.
- Deja en manos de la implementación de JPA la selección de una estrategia de generación dependiendo de la base de datos (el dialecto configurado en Hibernate).
- Ejemplo:

```
@Entity
public class Student {

   @Id
   @GeneratedValue
   private long studentId;
}
```

AUTO

- Para usarla es imprescindible conocer cómo la ha implementado nuestro proveedor de JPA. En Hibernate este es el comportamiento:
  - MySQL\MariaDB pre 10.3: AUTO_INCREMENT con Hibernate 4, TABLE a partir de Hibernate 5.
  - MariaDB 10.3: una única secuencia (hibernate_sequence).
  - PostgreSQL: una única secuencia (hibernate_sequence).
  - Oracle: una única secuencia (hibernate_sequence).
  - SQL Server: IDENTITY.
- Obsérvese que es **una mala idea usar este generador** si nuestro software se utiliza con **MySQL** o en **versiones antiguas de MariaDB** porque Hibernate aplica el tipo **TABLE** que, tal y como hemos visto, debemos evitar. Tampoco es una elección afortunada en el caso de **SQL Server**: si bien IDENTITY es preferible a TABLE, en SQLServer 2012 tenemos disponible SEQUENCE que es mejor todavía.

AUTO

- Si usamos el tipo de generación por defecto, el proveedor de persistencia determinará los valores según el tipo del atributo de clave principal. Este tipo puede ser **numérico o UUID**.

- Para **valores numéricos**, la generación se basa en un generador de **secuencias o tablas**, mientras que los **valores de UUID** utilizarán el **UUIDGenerator**.

```
@Entity
public class Student {

   @Id
   @GeneratedValue
   private long studentId;
}
```

- En este caso, los valores de la clave principal serán únicos a nivel de la base de datos.

AUTO

- Ahora veremos el **UUIDGenerator**, que se introdujo en Hibernate 5.
- Para usar esta característica, solo necesitamos declarar **un identificador de tipo UUID** con la **anotación @GeneratedValue**:

```
@Entity
public class Course {

   @Id
   @GeneratedValue
   private UUID courseId;
}
```

- Hibernate generará una identificación de la forma "8dd5f315-9788-4d00-87bb-10eed9eff566".

AUTO

- La **versión 3.1 de JPA** incorpora a la especificación varias novedades. La que nos interesa es la posibilidad de usar la **clase java.util.UUID** para declarar cualquier atributo persistible de una clase de tipo entidad, incluyendo el identificador.

- UUID significa **«identificador único universal»**, definición estupenda porque no deja mucho a la imaginación. Un UUID pretende identificar un mismo elemento de forma única en sistemas distintos y, quizás, en cualquier sistema existente (**la probabilidad de duplicidad es baja**).

- Otra de las fortalezas reseñables de los UUID es que pueden ser generados de manera independiente por cualquier sistema sin requerirse de un servicio central que los proporcione. En la era de los **microservicios/sistemas distribuidos** esta característica es muy interesante.

- El concepto UUID toma cuerpo en un valor de 128 bits representado visualmente como 36 caracteres alfanuméricos separados en cinco grupos. Aquí tienes una muestra:

```
dd180102-b094-11ed-afa1-0242ac120002
```

AUTO

- El ejemplo expone el principal punto débil de los UUID: **su exagerado tamaño** para tratarse de una **clave primaria subrogada**.
- Un UUID consume un **gran espacio** en la base de datos, en concreto el doble de los 64 bits de los enteros grandes que vimos. Esto no solo afecta a la tabla con la clave, sino también a aquellas que la referencien como clave ajena.
- Otro **inconveniente** a considerar, de suma importancia, es que el **índice para la clave puede tener un rendimiento pobre** en ciertas bases de datos debido al carácter pseudoaleatorio y desordenado de los valores de las claves. Por estos motivos, **solo se recomienda utilizar UUIDs cuando sea imprescindible**.
- Hibernate admite UUID como identificadores desde hace años y ahora, como dije, esta capacidad ha sido estandarizada por JPA 3.1. Lo siguiente es lícito:

```
@Id
private UUID uuid;
```

AUTO

- ¿Qué columna requiere **uuid**?
- Podemos guardar la representación visual en un CHAR (36) o equivalente, pero lo eficiente es guardarlo en un formato binario o bien un tipo uuid si la base de datos ofrece esta opción.
- Esta tabla recoge el tipo idóneo en las bases de datos:
  - MySQL \MariaDB: BINARY(16).
  - PostgreSQL: uuid.
  - SQL Server: uniqueidentifier.
  - Oracle: RAW(16).

AUTO

- ¿Cómo creamos un UUID válido?
- Java cuenta con el método **UUID#randomUUID** y algunas bases de datos proveen funciones generadoras de UUIDs.
- Pero para nosotros lo más conveniente es utilizar el generador GenerationType.UUID y olvidarnos del asunto:

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

AUTO

- JPA exige a su proveedor calcular un UUID que cumpla con el estándar RFC 4122.
- Pero hay un problemilla: no indica cuál de las cinco técnicas de cálculo propuestas por el RFC debe aplicarse.
- Hibernate opta por la **denominada versión 4**, fundamentada en **números pseudoaleatorios**.
- Como alternativa, puedes solicitar a Hibernate que elija la versión 1 del RFC basada en un **timestamp** y la **dirección MAC**, aunque Hibernate usa la IP en lugar de la MAC:

```
@Id
@GeneratedValue
@UuidGenerator(style = UuidGenerator.Style.TIME)
private UUID id;
```

Identificadores compuestos

- Introducción.
- La anotación @Embeddable.
- La anotación @EmbeddedId.
- La anotación @IdClass.

Introducción

- Lo más práctico es usar **una clave subrogada**.
- Con todo, ¿qué pasaría en JPA si nuestra clave primaria fuera múltiple (más de una columna)?
- Es el caso en que nos encontraremos si tenemos **una relación m:n y queremos** (o necesitamos) modelar **con una clase la tabla intermedia** con una clase asociación.
- También tendremos este problema si debemos **adaptarnos a bases de datos ya existentes** que empleen este tipo de claves.

Introducción

- Además de los **identificadores simples** que hemos visto hasta ahora, Hibernate también nos permite definir **identificadores compuestos**.
- Un **identificador compuesto** está representado por una **clase que define la clave principal** con uno o más atributos persistentes. La **clase de la clave principal** debe cumplir varias condiciones:
  - Debe definirse mediante **anotaciones @EmbeddedId o @IdClass**.
  - Debe ser **pública**, **serializable** y tener un **constructor público sin argumentos**.
  - Finalmente, debería implementar los **métodos equals() y hashCode()**.
- Los atributos de la clase pueden ser básicos, compuestos o ManyToOne, evitando colecciones y atributos OneToOne.

La anotación @Embeddable

- Ahora veamos cómo definir una identificación usando @EmbeddedId. Primero, necesitamos una **clase de clave principal** anotada con @Embeddable:

```
@Embeddable
public class OrderEntryPK implements Serializable {

    private long orderId;
    private long productId;

    // standard constructor, getters, setters
    // equals() and hashCode()
}
```

La anotación @EmbeddedId

- A continuación, podemos agregar **una identificación de tipo OrderEntryPK** a una entidad usando **@EmbeddedId**:

```
@Entity
public class OrderEntry {

   @EmbeddedId
   private OrderEntryPK entryId;

   // ...
}
```

La anotación @IdClass

- La **anotación @IdClass** es similar a **@EmbeddedId**. La **diferencia** con @IdClass es que los **atributos se definen en la clase de entidad principal** usando **@Id para cada uno**.
- La **clase de clave principal** tendrá el **mismo aspecto que antes**.
- El ejemplo de OrderEntry con un @IdClass sería el siguiente:

```
@Entity
@IdClass(OrderEntryPK.class)
public class OrderEntry {
   @Id
   private long orderId;
   @Id
   private long productId;

}
```