Les clés primaires avec JPA

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 Concepts fondamentaux 1
2 Génération automatique des Id atomiques 1
2.1 Tableau comparatif 2
2.2 Les générateurs SEQUENCE et IDENTITY 3
2.3 Sequence et Batch 4
2.4 générateur TABLE 4
2.5 UUIDs 5
3 Les clés composites 6
3.1 Embedded Id 6
3.2 Id Class 7

1 Concepts fondamentaux

- Clé primaire
  - sous-ensemble minimal K d'attributs A = {a_1, a_2, ..., a_n} d'une relation R qui satisfait simultanément :
    1. l'unicité, qui garantit que pour tous tuples t_1, $t_2 \in R$, si $t_1 \neq t_2$ alors leurs projections sur K sont différentes ($t_1[K] \neq t_2[K]$),
    2. la minimalité, qui assure qu'aucun sous-ensemble propre $K' \subset K$ ne peut satisfaire la propriété d'unicité,
    3. la non-nullité, qui impose que pour tout tuple $t \in R$ et tout attribut $k \in K$, la valeur t[k] ne peut pas être nulle ($t[k] \neq null$),
- Types supportés: primitifs (`int`, `long`, ...), wrappers (`Integer`, `Long`, ...), `String`, Date (`Date`, `LocalDate`, ...), `UUID`
- Annotation `@Id` sur l'attribut de l'entité

2 Génération automatique des Id atomiques

En théorie des bases de données relationnelles, une clé primaire est définie comme un sous-ensemble minimal K d'attributs A = {a_1, a_2, ..., a_n} d'une relation R qui satisfait

simultanément trois propriétés fondamentales : (1) l'unicité, qui garantit que pour tous tuples t_1, $t_2 \in R$, si $t_1 \neq t_2$ alors leurs projections sur K sont différentes ($t_1[K] \neq t_2[K]$), (2) la minimalité, qui assure qu'aucun sous-ensemble propre $K' \subset K$ ne peut satisfaire la propriété d'unicité, et (3) la non-nullité, qui impose que pour tout tuple $t \in R$ et tout attribut $k \in K$, la valeur t[k] ne peut pas être nulle ($t[k] \neq null$), ces propriétés garantissant ainsi l'intégrité des entités et l'identification unique des tuples dans la base de données.

La génération automatique des identifiants est une fonctionnalité essentielle de JPA, permettant de déléguer la création des valeurs d'identifiants au framework.

Plusieurs stratégie de génération sont disponibles, chacune adaptée à un contexte particulier.

1. **IDENTITY**
   - Utilise l'auto-increment natif de la base
   - Optimal pour MySQL/MariaDB
   - Pas de batch possible
2. **SEQUENCE**
   - Utilise une séquence en base
   - Performant avec allocationSize > 1
   - Standard pour PostgreSQL/Oracle
3. **TABLE**
   - Utilise une table dédiée
   - Portable mais moins performant
   - Solution de dernier recours
4. **AUTO**
   - Choix automatique selon la base
   - IDENTITY pour MySQL
   - SEQUENCE pour PostgreSQL
5. **UUID**
   - Utilise un UUID
   - Portable et unique

2.1 Tableau comparatif

Stratégie	Performance	Portabilité	Batch	Scalabilité	Usage recommandé
IDENTITY	+++	+	Non	+	MySQL/MariaDB
SEQUENCE	++	++	Oui	++	PostgreSQL/Oracle
TABLE	+	+++	Oui	+++	Multi-bases

Stratégie	Performance	Portabilité	Batch	Scalabilité	Usage recommandé
AUTO	++	++	Selon	++	Développement
UUID	+	+++	Non	++++	Systèmes distribués

Notes sur la scalabilité:

- IDENTITY: problématique en sharding
- SEQUENCE: possible avec sequences par shard
- TABLE: adapté au clustering
- UUID: optimal pour systèmes distribués

2.2 Les générateurs SEQUENCE et IDENTITY

- Les générateurs SEQUENCE et IDENTITY spécifient l'utilisation d'une séquence de base de données ou d'une colonne d'identité (plus performant, mais pas portable). .

```
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@AllArgsConstructor
@Builder
@Getter

@Entity(name = "CustomerSeqId")
@Table(name="CUSTOMER", schema = "EX_IDSEQ")
public class Customer {
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"seq_customer")
@SequenceGenerator(name = "seq_customer", sequenceName = "CUSTOMER_SEQUENCE",
allocationSize = 10)
private Long id;

private String name;
}
```

Figure 1. – EmployeePK.java

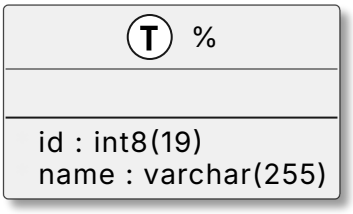


Figure 2. – An Employee with a composite Key/Id

2.3 Sequence et Batch

La génération de clés avec une séquence peut être optimisée en utilisant un `allocationSize` plus grand que 1. Cela permet de pré-allouer un bloc de valeurs pour les entités, réduisant ainsi le nombre d'appels à la séquence.

```
// Exemple de configuration optimisée
@Entity
public class Order {
    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "order_seq"
    )
    @SequenceGenerator(
        name = "order_seq",
        sequenceName = "ORDER_SEQ",
        initialValue = 1,
        allocationSize = 50  // Optimisation batch
    )
    private Long id;
}
```

2.4 générateur TABLE

Le générateur TABLE demande d'attribuer des clés primaires à l'entité en utilisant une table de base de données, c'est le plus portable mais le moins performant.

```java
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@AllArgsConstructor
@Builder
@Getter

@Entity(name = "CustomerTableId")
@Table(name="CUSTOMER", schema = "EX_IDTABLE")
public class Customer {
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator =
"customer_table_generator")
@TableGenerator(name = "customer_table_generator", schema = "EX_IDTABLE",
table = "ID_GENERATOR_TABLE")
private Long id;

private String name;
}
```

Figure 3. – EmployeePK.java

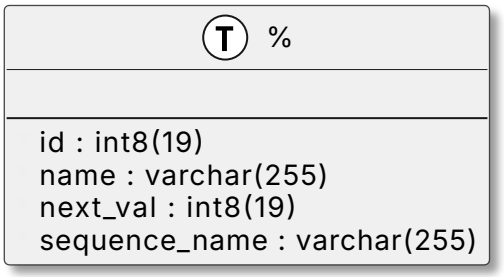


Figure 4. – An Employee with a composite Key/Id

2.5 UUIDs

- Les UUIDs sont des identifiants universels uniques, générés aléatoirement.
- Avantages: unicité globale, distribution, sans coordination centrale
- Inconvénients: taille (16 octets), performance index, non séquentiel
  - Pour la pagination ajouter un index sur un attribut date de création
- Usage: microservices, systèmes distribués, réplication multi-maîtres

```
@Getter
@Setter
@ToString
@RequiredArgsConstructor(staticName = "of")
@NoArgsConstructor(access = AccessLevel.PROTECTED)

@Entity
@Table(name = "CUSTOMER", schema = "EX_UUID")
public class Customer {
@Id
@GeneratedValue(strategy = GenerationType.UUID)
private UUID uuid;

@Column(length = 50, nullable = false)
@NonNull
private String name;
}
```

Figure 5. – EmployeePK.java

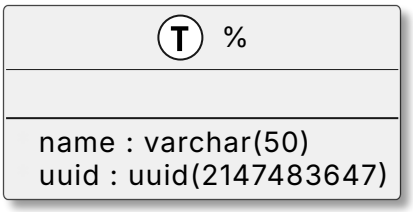


Figure 6. – A Customer with a UUID

3 Les clés composites

- Types de clés
  - **Clé atomique**
    - **Natural** (Clé naturelle) : attribut métier unique (email, SIRET)
    - **Surrogate** (Clé artificielle) : identifiant technique unique (Long, UUID)
  - **Clé composite**
    - **Natural** : combinaison d'attributs métier (année + département + matricule)
    - **Surrogate** : combinaison d'identifiants techniques (shardId + sequenceId)
- JPA propose deux approches pour gérer les clés composites:
  - `@EmbeddedId` : encapsule les attributs dans une classe dédiée
  - `@IdClass` : distribue les attributs dans l'entité

3.1 Embedded Id

- Utilisation de l'annotation `@EmbeddedId`

- Encapsulation des attributs de la clé dans une classe dédiée
- La classe embarquée doit être annotée `@Embeddable`
  - Doit implémenter Serializable
  - Nécessite equals() et hashCode()

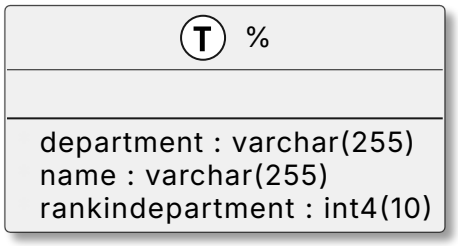


Figure 7. – An Employee with a composite Key/Id

```
@Getter
@Setter

@Entity(name="EmployeEmbeddedId")
@Table(name="EMPLOYE", schema = "EX_EMBEDDED_ID")
public class Employee {
@EmbeddedId
private EmployeePK id;
private String name;
}
```

Figure 8. – Employee.java

```
@EqualsAndHashCode
@Embeddable
@AllArgsConstructor
@NoArgsConstructor
public class EmployeePK implements Serializable {
private String department;
private int rankInDepartment;
}
```

Figure 9. – EmployeePK.java

3.2 Id Class

- Utilisation de l'annotation `@IdClass` dans l'entité.
- Les attributs de la clé sont définis dans l'entité
- Une classe séparée définit la structure de la clé
- Les noms des attributs doivent correspondre
- Plus simple pour les requêtes JPQL

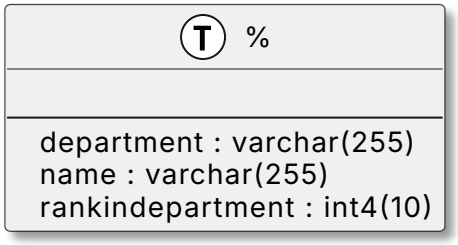


Figure 10. – An Employee with a composite Key/Id

```
@Setter
@Getter

@Entity(name="EmployeIdClass")
@Table(name="EMPLOYE", schema = "EX_IDCLASS")
@IdClass(EmployeePK.class)
public class Employee {
@Id
private String department;
@Id
private int rankInDepartment;
private String name;
}
```

Figure 11. – Employee.java

```
@EqualsAndHashCode
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class EmployeePK implements Serializable {
@Id
private String department;
@Id
private int rankInDepartment;
}
```

Figure 12. – EmployeePK.java