R6.05 - Prog. Frameworks : Spring boot

TD n°3 : mapping relationnel

Détails
Écrit par stéphane Domas
Catégorie : R6.05 - Prog. Framework : Spring boot (/index.php/s6/objets-connectes)
Publication : 31 décembre 2020
Affichages : 568

Préambule

Les TD 1 & 2 se basent sur une seule table en BdD et le TP 1 utilise des jointure faites "à la main" entre 2 tables. Dans les faits, un modèle relationnel correct utilise le principe de clef étrangère pour établir des relations entre des tables. Le problème est de traduire cette relation en une relation entre objets puisque dans les projets spring boot, on manipule des instances plutôt que de traiter directement des enregistrements en BdD. Pour cela, hibernate propose un ensemble d'annotations que l'on utilise dans les entités, la difficulté étant de bien choisir quelle annotation utiliser pour représenter quelle relation.

Le projet de démonstration des points abordés est téléchargeable [ici (/upload/supports/S6/SpringBoot/TD/td3/springboot-td3-src.tgz)]

1°/ Les mappings relationnels avec hibernate

1.1°/ Kesako

La notion de mapping relationnel vient du fait que l'on cherche à traduire (donc "mapper") une relation dans un modèle de BdD relationnel en une relation entre des classes, à savoir des relation du type :

- 1, 1 : un élément d'une table est en relation avec un seul autre élément d'une table (par ex, table `user` en relation avec table `user_profile` = un utilisateur n'a qu'un seul profil associé).
- 1, * : un élément d'une table est en relation avec plusieurs autres éléments d'une table (par ex, table `user` en relation avec table `user_bank_account` = un utilisateur peut avoir plusieurs comptes bancaires)
- *, * : plusieurs éléments d'une table se trouvent en relation avec plusieurs éléments d'une autre table (par ex, table `user` en relation avec table `rooms` = un utilisateur utilise plusieurs salles & une salle est utilisée par plusieurs utilisateurs)

Or, l'expressivité des classes est plus grande que celle d'un modèle relationnel, il est possible de décrire ces 3 types par plus que 3 solutions. Prenons un exemple pour le montrer. Soit une table `user` et une table `user_bank_account` avec une relation de type 1,*. Cela implique que la table `user_bank_account` contient un clef étrangère correspondant à la clef primaire de `user`.

Si on doit modéliser cette relation en objet, on a globalement 3 choix :

1. la classe `User` ne contient rien permettant de le raccrocher à un compte bancaire et la classe `UserBankAccount` contient un attribut de type `User`
2. la classe `UserBankAccount` ne contient rien permettant de le raccrocher à un utilisateur et la classe `User` contient un attribut de type `List<UserBankAccount>`
3. la classe `User` contient un attribut de type `List<UserBankAccount>` et la classe `UserBankAccount` contient un attribut de type `User`

La solution 1 est celle qui se rapproche le plus du modèle relationnel, alors qu'il est strictement impossible de représenter les 2 et 3 en SQL. Pourtant, la dernière solution serait sans doute la plus pratique pour créer une application bancaire. En effet, quand on édite un utilisateur, on accède directement à ses comptes, et quand on édite un compte, on retrouve directement son propriétaire.

Cela dit, la solution 3 n'est pas "forcément" souhaitable. Par exemple, dans un bibliothèque, on cherche généralement des livres, plutôt que des auteurs. On aurait donc plutôt besoin d'une solution 1, avec la classe `Book` qui contient un attribut de type `Author`. Sur un blog, un article peut recevoir des commentaires, mais on ne cherche jamais un commentaire en particulier. On aruait donc une solution 2, avec la classe `Article` qui contient un attribut de type `List<Comment>`.

De fait, il serait intéressant de pouvoir utiliser l'une de ces 3 solutions au choix selon l'application, alors qu'au niveau BdD, les tables seraient exactement les mêmes.

1.2°/ les bases

Pour définir un mapping relationnel classique, hibernate propose des annotations que l'on place dans les classes entités, devant les attributs représentant d'autres entités. Il y a 4 annotations principales :

- `@OneToOne` : mapping d'une relation 1,1
- `@OneToMany` et `@ManyToOne` : mapping d'une relation 1,*
- `@ManyToMany` : mapping d'une relation *,*.

Selon les cas, on doit également utiliser :

- des paramètres à ces annotations (`mappedBy`, `cascade`, `nullable`, ...)
- d'autres annotations, notamment `@JoinColumn` et `@JoinTable`, pour préciser quelles colonnes/table on utilise dans le modèle relationnel.

L'emplacement de ces annotations et leur paramètres dépend du modèle objet que l'on veut obtenir, tout en restant cohérent avec le modèle relationnel. Dans tous les cas, il est possible de créer une relation réflexive ou non, ce qui se traduit dans le jargon hibernate par bidirectionnelle (= réflexive) ou unidirectionnelle (= non réflexive). L'emplacement des annotations et leurs paramètres va également imposer un côté propriétaire ("owner") et un côté possédé ("owned"), ce qui aura des conséquences lors de certaines opération en BdD.

Comme il y a 3 types de relations, 4 annotations et 2 "sens", on aboutit à un ensemble fini et relativement simple de solutions, qui sont résumées dans les 2 tableaux suivants. Ces tableaux utilisent des exemples "pratiques" de tables et de classes pour être plus parlant, mais ils se basent sur le fait qu'il existe des classes entités `ObjA` et `ObjB`, représentant les tables `table_a` et `table_b`, avec une relation entre les deux du type :

- 1,1 = un seul élément de `table_b` référence un seul élément de `table_a` = une clef étrangère dans `table_b` correspond à la clef primaire de `table_a`.
- 1,* = plusieurs éléments de `table_b` référencent un seul élément de `table_a` = une clef étrangère dans `table_b` correspond à la clef primaire de `table_a`.
- *,* = plusieurs éléments de `table_b` référencent plusieurs éléments de `table_a` = une table table_a_b contient un couple de clef étrangères correspondant aux clefs primaires de `table_a` et `table_b`.

Unidirectionnel

1,1

```java
@Entity
@Table(name="user_profiles")
class UserProfile {
  @Id
  private Long id;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="users")
class User {
  @Id
  private Long id;
  @OneToOne  // this class is the owner
  // users has a foreign key user_profile_id, refering to primary key of user_profiles
  @JoinColumn(name="user_profile_id")
  private UserProfile userProfile;
  ... // other attributes, getters/setters, ...
}
```

1,*

solution avec `@OneToMany`

```java
@Entity
@Table(name="bank_users")
class BankUser {
  @Id
  private Long id;
  @OneToMany  // this class is the owner
  // !!! bank_accounts has a foreign key user_id, refering primary key of bank_users !!!
  @JoinColumn(name="bank_user_id")
  private List<BankAccount> bankAccounts;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="bank_accounts")
class BankAccount {
  @Id
  private Long id;
  ... // others attributes, getters/setters, ...
}
```

solution avec `@ManyToOne`

```java
@Entity
@Table(name="bank_users")
class BankUser {
  @Id
  private Long id;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="bank_accounts")
class BankAccount {
  @Id
  private Long id;
  @ManyToOne  // this class is the owner
  // bank_accounts has a foreign key bank_user_id, refering to primary key of bank_users
  @JoinColumn(name = "bank_user_id")
  private BankUser bankUser;
  ... // others attributes, getters/setters, ...
}
```

,

```java
@Entity
@Table(name="students")
class Student {
  @Id
  private Long id;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="courses")
class Course {
  @Id
  private Long id;
  @ManyToMany  // this class is the owner
  @JoinTable(name = "courses_students",
             joinColumns = @JoinColumn(name = "course_id"),
             inverseJoinColumns = @JoinColumn(name = "student_id"))
  private List<Student> students;
  ... // others attributes, getters/setters, ...
}
```

Remarque : pour la relation 1,1, il est parfaitement possible d'inverser la relation et d'avoir la table `users` qui contient une clef étrangère correspondant à la clef primaire de `user_profiles`. Le choix entre les deux solutions dépend de la façon de modéliser la BdD en tenant compte des contraintes fonctionnelles (pratiques métiers, quelle table est accédée en priorité, ...)

Bidirectionnel

1,1

```java
@Entity
@Table(name="user_profiles")
class UserProfile {
  @Id
  private Long id;
  @OneToOne(mappedBy="userProfile") // user = name of the attribute in class User
  private User user;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="users")
class User {
  @Id
  private Long id;
  @OneToOne  // this class is the owner
  // users has a foreign key user_profile_id, refering to primary key of user_profiles
  @JoinColumn(name="user_profile_id")
  private UserProfile userProfile;
  ... // other attributes, getters/setters, ...
}
```

1,*

```java
@Entity
@Table(name="bank_users")
class BankUser {
  @Id
  private Long id;
  @OneToMany(mappedBy="bankUser") // bankUser = name of attribute in class BankAccount
  private List<BankAccount> bankAccounts;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="bank_accounts")
class BankAccount {
  @Id
  private Long id;
  @ManyToOne  // this class is the owner
  // bank_accounts has a foreign key bank_user_id, refering to primary key of bank_users
  @JoinColumn(name = "bank_user_id")
  private BankUser bankUser;
  ... // others attributes, getters/setters, ...
}
```

,

```java
@Entity
@Table(name="students")
class Student {
  @Id
  private Long id;
  @ManyToMany(mappedBy="students") //  students = name of attribute in class Course
  private List<Course> courses;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="courses")
class Course {
  @Id
  private Long id;
  @ManyToMany  // this class is the owner
  @JoinTable(name = "courses_students",
          joinColumns = @JoinColumn(name = "course_id"),
          inverseJoinColumns = @JoinColumn(name = "student_id"))
  private List<Student> students;
  ... // others attributes, getters/setters, ...
}
```

Toutes ces situations (excepté 1,1 bidirectionnel qui n'est jamais très utile) sont illustrées dans le projet de démonstration.

1.3°/ Cas du *,* avec données associées

Dans les tableaux ci-dessus, les relations de type *,* simples sont traduite avec `@ManyToMany`. Cependant, il est fréquent en BdD que la table intermédiaire qui contient le couple de clefs étrangère serve également à stocker des valeurs. Un exemple classique est celui d'un étudiant suivant un cours, pour lequel il est évalué. Il y a une relation de type *,* entre les tables étudiants et cours, mais il faut bien stocker les notes quelque part. Ce problème se résout en stockant la note dans la troisième table qui contient le couple de clef étrangère.

Même si d'un point de vue BdD, cela reste une relation *,*, il n'est plus possible d'utiliser `@ManyToMany` pour la traduire en objet. Une des façons les plus simples et élégante de résoudre ce problème avec hibernate consiste à :

- utiliser le principe de classe embarquée (= embedded) pour représenter le couple de clefs étrangères,
- créer une classe entité pour représenter la table intermédiaire, dont l'id sera du type de la classe embarquée,
- utiliser des relation de type 1,* entre la table intermédiaires et les 2 autres, bidirectionnelles ou non.

Par exemple, avec le même exemple students/courses de la section 1.2, auquel on ajoute que l'on veut stocker dans la table intermédiaire `courses_students` un nombre à virgule `rank`, et que la classe représentant cette table est nommée `CourseStudent`, alors on doit écrire 4 classes, comme suivant :

- la classe embarquée, qui doit utiliser l'annotation `@Embeddable` et implémenter l'interface `Serializable` :

```java
@Embeddable
class CourseStudentKey implements Serializable {
   @Column(name="course_id") // name of the foreign key in courses_students, refering to primary key of courses
   private Long courseId;
   @Column(name="student_id") // name of the foreign key in courses_students, refering to primary key of students
   private Long studentId;
  ... // constructor, getters/setters
}
```

- la classe représentant la table `courses_students` qui doit utiliser `@EmbeddedId` pour annoter l'attribut représentant l'objet clef embarqué, ainsi que `@ManyToOne` et `@MapsId` devant les attributs de type `User` et `Student`.

```java
@Entity
@Table(name="courses_students")
class CourseStudent {
  @EmbeddedId
  private CourseStudentKey courseStudentKey;

  @ManyToOne
  @MapsId("courseId") // courseId = name of the attribute in the embedded id
  // courses_students contains a foreign key course_id refering to primary key of courses
  @JoinColumn(name = "course_id")
  private Course course;

  @ManyToOne
  @MapsId("studentId") //studentId = name of the attribute in the embedded id
  // courses_students contains a foreign key student_id refering to primary key of students
  @JoinColumn(name = "student_id")
  private Student student;

  // the value to store
  private double rank;
  ... // constructors, getters/setters
}
```

- les classes représentant les tables `courses` et `student`, comme montré en section 1.2 pour des relations 1,*. Par exemple, si on opte pour un mapping unidirectionnel pour `Student` et bidirectionnel pour `Course`, cela s'écrit :

```java
@Entity
@Table(name="students")
class Student {
  @Id
  private Long id;
  ... // other attributes, getters/setters, ...
}

@Entity
@Table(name="courses")
class Course {
  @Id
  private Long id;
  @OneToMany(mappedBy="course") // course = name of attribute in class CourseStudent
  private List<CourseStudent> coursesStudents;
  ... // other attributes, getters/setters, ...
}
```

1.4°/ Récupérer des objets

1.4.1°/ le principe par défaut

Grâce au mapping relationnel et aux repositories, il n'y a (quasi) jamais besoin d'écrire des requêtes SQL avec jointure pour obtenir des objets comportant des données venant de plusieurs tables. Hibernate et spring se débrouillent pour créer toutes les instances de classes nécessaires. Prenons par exemple le cas d'un mapping one-to-one bidirectionnel entre `ObjA` et `ObjB` pour illustrer ce qui se passe.

A priori, on va créer deux classes de type `JpaRepository` pour faire des transactions sur les tables `table_a` et `table_b`. On peut ainsi utiliser la méthode `findById()` pour récupérer une instance de `ObjA` (ou `ObjB`). Mais comme il existe un mapping, hibernate va en réalité lancer deux requêtes :

- une pour récupérer l'enregistrement demandé dans `table_a`,
- une pour récupérer l'enregistrement dans `table_b` avec une clef étrangère égale à la clef primaire de l'enregistrement récupéré juste avant.

Ensuite, hibernate crée une instance de `ObjA` et `ObjB`, puis met à jour leur attribut a et b. On obtient une référence croisée entre objet : ils se connaissent l'un l'autre. A noter que si le mapping était unidirectionnel, on aurait seulement le côté propriétaire qui aurait une référence vers l'autre.

1.4.2°/ Le problème principal

Ce principe s'applique pour les autres types de mapping excepté que hibernate va devoir créer parfois des collections d'objet. Dans ce cas, si on laisse spring boot et hibernate faire la travail de façon automatique, on risque de lancer énormément de requêtes et pas forcément de façon judicieuses.

Prenons l'exemple des étudiants assistant à des cours, avec un simple mapping many-to-many bidirectionnel entre les classes `Etudiant` et `Cours`, plus un `JpaRepository` pour chacune. Si on appelle findAll() pour récupérer tous les étudiants, voici ce que hibernate va faire :

- il récupère tout le contenu de la table etudiants et cours, ce qui lui permet de récupérer les ids de chaque étudiant,
- pour chaque id étudiant, il récupère les couples etudiant/cours dans la table intermédiaire, et grâce à une jointure de récupérer également les informations des cours auxquels assiste l'étudiant,

Cela veut donc dire que les informations de chaque cours vont être récupérées à priori autant de fois qu'il y a d'étudiant dans ce cours, ce qui n'est pas forcément très efficace.

Par exemple, s'il y a 10 étudiants assistant tous à 6 cours, il y aura quand même 11 requêtes :

- 1 requête pour obtenir tous les étudiants,
- 10 requêtes pour obtenir les couples étudiants/cours et les informations de chaque cours auquel assiste chaque étudiant.

Pour résoudre ce problème, il y a deux approches :

- ne pas mettre en place une relation bidirectionnelle entre étudiant et cours (cf. exemple many-to-many dans le tableau unidirectionnel)
- utiliser le mécanisme de "récupération paresseuse" (= lazy fetching) de hibernate.

La première approche est simple mais elle complique la tâche de récupération des cours pour un étudiant donné, puisque c'est au développeur d'écrire du code pour gérer cette opération (c.a.d un repository et une fonction de service). La deuxième approche est tout aussi simple et ne demande aucun code supplémentaire. Pour ce faire, on utilise le paramètre d'annotation `fetch`.

Par exemple, si on ne veut pas forcément charger les informations des cours d'un étudiant, on écrit la classe `Etudiant` comme suivant :

```java
@Entity
@Table(name="etudiants")
class Etudiant {
  @Id
  private Long id;
  ...
  @ManyToMany (mappedBy="etudiants", fetch = FetchType.LAZY),
  private List<Cours> cours;
  ...
}
```

Si on récupère un objet `Etudiant`, son attribut `cours` ne sera pas réellement initialisé avec les valeurs venant de la BdD. Cela restera le cas tant que l'on manipule uniquement les autres attributs de la classe. En revanche, dès que l'on va accéder aux éléments de la liste cours, hibernate fera la requête pour aller chercher les informations manquantes. ATTENTION, quand un contrôleur doit renvoyer un objet pas complètement initialisé, spring va par défaut compléter son initialisation avant de le traduire en JSON.

A noter que l'on pourrait également utiliser le lazy fetching dans la classe `Cours` pour éviter de charger les informations des étudiants assistant à un cours.

En conclusion, il est souvent recommandé d'utiliser le lazy fetching pour éviter trop de requêtes inutiles.

1.5°/ Cascade d'opérations

Le fait de créer, supprimer, mettre à jour des objets qui contiennent potentiellement des données associées à plusieurs tables pose un problème à hibernate : est-ce que l'on applique ces opérations seulement à la table associée à l'objet principal, à toutes les tables concernées, ou encore une autre solution.

Par défaut, hibernate utilise la première solution, ce qui peut provoquer des erreurs lorsque l'on veut sauver des objets en BdD. Exemple de code erroné dans un service, avec la relation 1,1 unidirectionnelle de la section 1.2 :

```java
UserProfile userProfile = new UserProfile(...);
User user = new User(userProfile, ...);
userRepository.save(user); // ERROR without cascading

userProfileRepository.save(userProfile); // OK but need to create repository for UserProfile :-(
userRepository.save(user); // OK even without cascading since profile has been saved in DB
```

Comme on le voit dans cet exemple, on est par défaut obligé de sauver les objets "internes", avant de pouvoir sauver l'objet principal. C'est une contrainte parfois utile mais pénible. C'est pourquoi hibernate permet de "cascader" à la demande les opérations. Pour cela, il faut utiliser le paramètre d'annotation cascade, avec comme valeur pour quelles opérations on veut cascader.

Par exemple, si on écrit `@OneToOne ( cascade = CascadeType.ALL)` dans la classe User, alors **toute opération** sur un objet `User` sera faite également sur l'objet "interne" `UserProfile`.

Il est cependant possible d'être plus précis. En effet, Hibernate définit différents types d'opérations sur les objets, 3 ayant un impact direct sur la BdD puisqu'elles implique des requêtes : création, mise à jour, suppression, qui correspondent respectivement aux valeurs :

- `CascadeType.PERSIST`,
- `CascadeType.MERGE`,
- `CascadeType.REMOVE`.

Il est donc possible de cascader uniquement pour la création/mise à jour, mais pas quand on supprime. C'est particulièrement utile avec certaines relations. Par exemple, quand on supprimer un client, on supprime également ses comptes bancaires. En revanche, on ne veut surtout pas supprimer un client si on supprimer l'un de ses comptes. Cela donne :

```java
@Entity
@Table(name="users")
class User {
  @Id
  private Long id;
  @OneToMany(mappedBy="user", cascade = CascadeType.ALL) // cascade for all operations
  private List<BankAccount> bankAccounts;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="bank_accounts")
class BankAccount {
  @Id
  private Long id;
  @ManyToOne (cascade = { CascadeType.PERSIST, CascadeType.MERGE})  // just cascade when creating/updating a bank account
  @JoinColumn(name = "user_id")
  private User user;
  ... // others attributes, getters/setters, ...
}
```

1.6°/ Sérialisation d'objets en JSON

Quand un contrôleur renvoie un objet, ce dernier est automatiquement "traduit" (= sérialisé) en un objet JSON équivalent. Le faut que cela soit automatique pose souvent problème lorsque les contrôleurs renvoient directement des objets entité issus de hibernate. En effet, s'il existe un mapping bidirectionnel entre deux entités, cela implique qu'elles contiennent chacune une référence vers l'autre. Dans ce cas, sérialiser l'un veut dire sérialiser l'autre, ce qui implique de re-sérialiser le premier, etc. On obtient donc une sérialisation récursive et infinie.

Même si la relation est unidirectionnelle, un mapping va provoquer la récupération de données potentiellement inutiles, ou non voulues immédiatement. Par exemple, on veut obtenir le profil d'un étudiant sans pour autant avoir les informations de tous les cours auxquels il participe. Dans ce cas, même une récupération de type lazy ne sera pas utile puisque les données non récupérées le seront quand même automatiquement au moment de la sérialisation.

Pour régler ces différents problèmes, il y a différentes solutions basées sur des annotations, plus ou moins versatiles :

- pour éviter une récursion : utiliser `@JsonManagedReference` du côté de l'entité principale, et `@JsonBackReference` du côté de l'entité encapsulée.
- pour "filtrer" la sérialisation en évitant de sérialiser certains attributs de l'entité encapsulée : utiliser `@JsonIgnoreProperties(...)`
- pour faire sa propre sérialisation de l'entité encapsulée : utiliser `@JsonSerialize(...)` (cf. démonstration demo2)

Bien entendu, la solution la plus souple, mais plus fastidieuse à écrire, consiste simplement à ne pas utiliser la sérialisation automatique des entités, et donc d'éviter que les contrôleurs ne renvoient des entités. Pour cela, il suffit de créer des classe de type DTO qui ne contiennent que ce que le client doit recevoir, construit à partir d'entités.

```java
@Entity
@Table(name="users")
class User {
  @Id
  private Long id;
  @OneToMany(mappedBy="user", cascade = CascadeType.ALL)
  @JsonManagedReference
  private List<BankAccount> bankAccounts;
  ... // other attributes, getters/setters, ...
}
```

```java
@Entity
@Table(name="bank_accounts")
class BankAccount {
  @Id
  private Long id;
  @ManyToOne (cascade = { CascadeType.PERSIST, CascadeType.MERGE})
  @JoinColumn(name = "user_id")
  @JsonBackReference // do not reserialize user
  private User user;
  ... // others attributes, getters/setters, ...
}
```

Exemple 1 : casser une récursion infinie de sérialisation

```java
@Entity
@Table(name="users")
class User {
  @Id
  private Long id;
  @OneToOne
  @JoinColumn(name="user_profile_id")
  @JsonIgnoreProperties({"password"}) // userProfile.password won't be serialized
  private UserProfile userProfile;
  ... // other attributes, getters/setters, ...
}
```

Exemple 2 : ignorer certains attributs lors de la sérialisation d'une entité encapsulée

```java
@Entity
@Table(name="bank_users")
class BankUser {
  @Id
  private Long id;
  String name;
  @OneToMany(mappedBy="bankUser") // bankUser = name of attribute in class BankAccount
  private List<BankAccount> bankAccounts;
}
```

```java
class BankUserDTO {
  Long id;
  String name;
  List<String> accountNumbers; // only send the account numbers, not the whole account infos
  public BankUserDTO(BankUser bankUser) {
    this.id = bankUser.id;
    this.name = bankUser.name;
    accountNumbers = new ArrayList<>();
    for(BankAccount a : bankUser.getBankAccounts()) {
      accountNumbers.add(a.getAccountNumber());
    }
  }
}
```

Exemple 3 : utiliser des DTO construits à partir d'entités

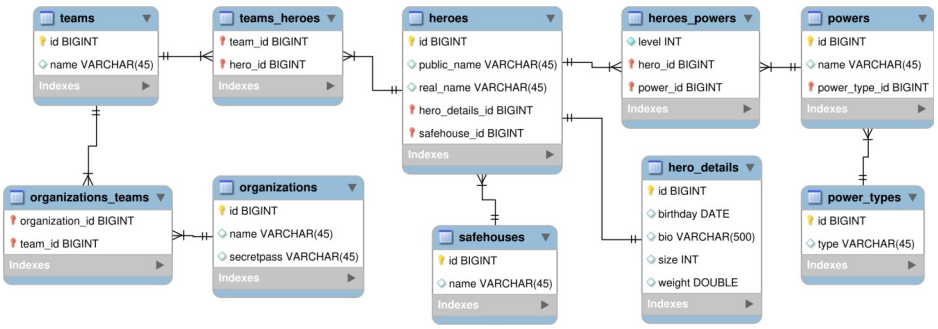
2°/ Projet de démonstration

L'archive téléchargeable du projet contient quatre modules, illustrant chacun un type de relation :

- demo1 : illustre une relation 1,1 entre un héros et son profil, mappée de façon one-to-one unidirectionnel,
- demo2 : illustre une relation 1,* entre un héros et des QGs mappée en one-to-many unidirectionnel, et entre des pouvoirs et des types de pouvoirs, mappée en many-to-one unidirectionnel,
- demo3 : illustre une relation *,*, entre des équipes et des héros, mappée en many-to-many unidirectionnel, et entre des organisations et des équipes, mappée en many-to-many bidirectionnel,
- demo4 : illustre une relation *,* avec valeurs, entre des héros et des pouvoirs + un niveau de pouvoir, mappé en many-to-one unidirectionnel côté héros, et many-to-one+one-to-many bidirectionnel côté pouvoirs.

ATTENTION : afin de rendre plus lisible et compréhensible chaque module, seuls les classes et attributs nécessaires à l'illustration sont utilisés dans le code,

Pour avoir malgré tout une vue d'ensemble, l'archive contient également un module demoAll synthétisant les 4 modules, afin de gérer le modèle relationnel complet donné ci-dessous



Précédent (/index.php/s6/objets-connectes/2583-td-n-6-fonctionnalites-avancees-acces-a-un-api-avec-axios-6)

Suivant (/index.php/s6/objets-connectes/2593-td-n-6-fonctionnalites-avancees-acces-a-un-api-avec-axios-11)

© 2025 cours-info

Haut de page