WILD CODE SCHOOL

TypeORM

Les relations

Les décorateurs de relations

Les types de relations

- TypeORM gère tous les types de relations avec des décorateurs :
  - @ManyToOne() : *Un élève à une école.*
  - @OneToMany() : *Une école à plusieurs élèves.*
  - @ManyToMany() : *Une école enseigne plusieurs langages.*
  - @OneToOne() : *Un élève a un profil.*

Les types de relations

- Chaque relation peut avoir des options :
  - **eager** : la relation sera chargée en même temps que l'entité
    - Attention, cela peut créer beaucoup de requêtes
  - **cascade** : les entités de la relations seront créés ou mises à jour en même temps que l'entité
  - **onDelete** : Supprime les entités reliées

https://typeorm.io/relations#relation-options

Many to One

Décorateur
@ManyToOne(options)

Un étudiant est dans une seule école.

On peut définir la relation inverse (OneToMany) dans le 2ème paramètre, ce n'est pas obligatoire.

Le décorateur **@ManyToOne** défini l'emplacement de la **clé étrangère**.

La liaison sera sauvegardée lors de l'appel de la méthode **save()**

src/entities/student.ts

```
import {
  …,
  ManyToOne
} from "typeorm";
import { School } from "./school";

@Entity()
export class Student extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 100 })
  firstname: string;

  @Column()
  @Index()
  name: string;

  …

  @ManyToOne(() => School, school => school.students)
  school: School;


}
```

One to Many

Décorateur
@OneToMany(options)

Une école à plusieurs étudiants.

Une école à alors un tableau d' étudiants.

Une **OneToMany ne peut fonctionner que si le côté ManyToOne est défini**. Ce dernier défini l'emplacement de la clé étrangère.

La liaison sera sauvegardée lors de l'appel de la méthode **save()**

src/entities/school.ts

```
import {
  BaseEntity,
  Column, Entity,
  PrimaryGeneratedColumn,
  Index,
  OneToMany
} from "typeorm";
import { Student } from "./student";

@Entity()
export class School extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  city: string;

  @Column()
  capacity: number;

  @OneToMany(() => Student, student => student.school)
  students: Student[];
}
```

Many to Many

Décorateur **@ManyToMany(options)**

Une école enseigne plusieurs langages.

Le décorateur **@JoinTable()** est obligatoire sur un des côté de la relation. Il défini la création d'une **table de jointure**.

Si un seul côté de la ManyToMany est défini, c'est une relation unidirectionnelle. A partir de l'entité, on peut récupérer la liste des langages enseignés mais pas l'inverse.

La liaison sera sauvegardée lors de l'appel de la méthode **save()**

src/entities/school.ts

```
import {
  …,
  ManyToMany,
  JoinTable
} from "typeorm";
import { Student } from "./student";
import { Language } from "./language";

@Entity()
export class School extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  …


  @ManyToMany(() => Language)
  @JoinTable()
  languages: Language[];
}
```

Many to Many

Définition bidirectionnelle.

Une many to many **peut se lire des 2 côtés**.

Une école enseigne plusieurs langages ET un langage est enseigné par plusieurs écoles.

Il est possible de récupérer en plus la liste des écoles enseignant un langage en définissant une **relation bidirectionnelle**.

Pas besoin de redéfinir le **@JoinTable()**, il ne doit être défini que d'un côté de la relation.

src/entities/school.ts

```
import {
  BaseEntity,
  Column, Entity,
  PrimaryGeneratedColumn,
  Index,
  OneToMany,
  ManyToMany,
  JoinTable
} from "typeorm";
import { School } from "./school";

@Entity()
export class Language extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  language: string;

  @ManyToMany(() => School, school =>
school.languages)
  schools: School[];
}
```

One to One

Un étudiant à un et un seul profil.

On définit **@JoinColumn()** du côté de la relation qui contiendra la **clé étrangère**.

Il peut être intéressant d'utiliser les options suivantes :

- **eager: true** : chargement automatique
- **cascade** : mise à jour et création en cascade
- **onDelete**: "CASCADE" : suppression en cascade

https://typeorm.io/one-to-one-relations

src/entities/school.ts

```
import {
  …
  OneToOne,
  JoinColumn
} from "typeorm";
import { School } from "./school";
import { Profile } from "./profile";

@Entity()
export class Student extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 100 })
  firstname: string;

  …

  @OneToOne(() => Profile)
  @JoinColumn()
  profil: Profile
}
```

Requêtage

Sur certaines ORM, les relations se chargent quand on accède à l'attribut. **Pas sur TypeORM.**

Chargement des entités au **find** ou **QueryBuilder**.

Sauf pour les relations en eager, nous allons devoir spécifier les relations à charger lors du requêtage.

Chargement des écoles ainsi que des étudiants et langages associés.

Exemple

```
const schools = await School.find({
    relations: {
      students: true,
      languages: true
    }
  });
```

Requêtes complexes

Les options de la méthode find permettent d'effectuer des requêtes complexes :

- **select** : sélection des attributs
- **where** : filtre des enregistrements
- **take** : équivalent à ***limit***
- **skip** : équivalent à ***offset***
- **order** : définition du tri

Exemple

```
import { MoreThan } from "typeorm";

await School.find({
    select: {
      id: true,
      city: true,
      address: true
    },
    relations: {
      students: true,
      languages: true
    },
    where: [{ capacity: MoreThan(30) }],
    order: {
      capacity: 'DESC'
    },
    skip: 10,
    take: 5
  });
```

Les migrations

Pour la production

Migrations

Désactivation de la synchronisation des modifications de la structure de la base.

Application incrémentale des modifications de la base.

En synchronisation, chaque modification d'entité est automatiquement reportée dans la base.

Sans la synchronisation, il faut générer un fichier de migration contenant tous les changements effectués puis l'exécuter.

Le fichier de migration peut contenir des requêtes de reprise de données.

Exécution des migrations à chaque déploiement en production.

Migrations : Typescript

Pour utiliser les migrations en TS, il faut installer **ts-node** en dépendance de développement :

```
$ npm install ts-node --save-dev
```

Nous allons également ajouter plusieurs **scripts** dans le package.json pour faciliter l'**utilisation des migrations**.

Les migrations seront contenues dans un **dossier migrations** que nous créerons à la racine du projet.

Migrations : Scripts

migration:create => Création d'une migration vide.
migration:generate => Génération d'un fichier de migration contenant les différences à appliquer.
migration:up => Exécution des migrations pas encore exécutées.
migration:down => Rollback de la dernière migration et ainsi de suite.
Ne pas oublier de définir l'emplacement de la datasource

package.json

```
{
  …
  "scripts": {
    …
    "migration:create": "typeorm-ts-node-commonjs migration:create",
    "migration:generate": "typeorm-ts-node-commonjs migration:generate -d
src/config/db.ts",
    "migration:up": "typeorm-ts-node-commonjs migration:run -d src/config/db.ts",
    "migration:down": "typeorm-ts-node-commonjs migration:revert -d src/config/db.ts"
  },
  …
}
```

Migrations : Datasource

Nous passons en **synchronise: false** pour ne plus appliquer automatiquement les changements apportés aux entités.

Nous définissons l'**emplacement des fichiers de migrations** dans le **dossier migrations** à la racine du projet.

Chaque migration exécutée est enregistrée dans une table. Seules les migrations absentes de cette table sont exécutées lors d'une nouvelle demande d'exécution.

src/config/db.ts

```
import { DataSource } from "typeorm";

export const dataSource = new DataSource({
  type: "sqlite",
  database: "./good_corner-typeorm.sqlite",
  entities: ["src/entities/*.ts"],
  synchronize: false,
  migrations: ["migrations/*.ts"],
  migrationsTableName: "migrations",
});
```

Migrations : Création

Création d'une **migration vide**.

Il faut définir l'emplacement du dossier de migration même si c'est redondant.

Une migration **implémente l'interface MigrationInterface**. Nous devons implémenter ces 2 méthodes :

- **up()** => SQL pour appliquer les changements
- **down()** => SQL pour rollback les changements

migrations/1694766115388-CreateMigrationSchool.ts

```
import { MigrationInterface, QueryRunner }
from "typeorm"

export class
CreateMigrationSchool1694766115388
implements MigrationInterface {

    public async up(queryRunner:
QueryRunner): Promise<void> {
    }

    public async down(queryRunner:
QueryRunner): Promise<void> {
    }

}
```

```
$ npm run migration:create migrations/CreateMigrationSchool
```

Migrations : Génération

Si on ne veut pas écrire la migration manuellement, on peut la **générer automatiquement**.

TypeORM compare l'état des entités par rapport à la structure de la base.

Génération d'un fichier contenant les requêtes SQL permettant de mettre la base à jour par rapport aux modifications apportées aux entités.

```
$ npm run migration:generate migrations/CreateStructure
```

Migrations : Génération

1694766695165-DeleteAddressFromStudent.ts

```
import { MigrationInterface, QueryRunner } from "typeorm";

export class DeleteAddressFromStudent1694766695165 implements MigrationInterface {
    name = 'DeleteAddressFromStudent1694766695165'

    public async up(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`DROP INDEX "IDX_eead2cd6e5be2c86303b786bff"`);
        …
    }

    public async down(queryRunner: QueryRunner): Promise<void> {
        await queryRunner.query(`DROP INDEX "IDX_eead2cd6e5be2c86303b786bff"`);
        await queryRunner.query(`ALTER TABLE "student" RENAME TO "temporary_student"`);
        …
    }

}
```

Migrations : Génération

Tout code généré doit être vérifié.

Soyez critique sur le code SQL généré. Vous pouvez le modifier si le code ne vous semble pas optimal cependant, le résultat final doit être le même.

Possibilité d'ajouter du code de reprise de données soit au format SQL soit en utilisant directement les entités TypeORM.

Migrations : Exécution / Rollback

Lancer les migrations:

```
$ npm run migration:up
```

Quand les lancer ?

En local, à chaque pull (s'il y en a une nouvelle).
En production, à chaque déploiement (tout le temps).

Possibilité d'annuler la dernière migration :

```
$ npm run migration:down
```

Atelier

The good corner

The Good Corner

En plus des catégories, nous aimerions pouvoir ajouter des tags aux annonces.

Une annonce peut avoir plusieurs tags ou aucun.

Un tag peut être lié à plusieurs annonces ou aucune.

Atelier : Catégorie : ManyToOne

- Créer l'entité Catégorie
- Renseigner l'entité dans la datasource
- Modifier le endpoint POST d'une annonce pour la lier à une catégorie
- Modifier le endpoint PUT d'une annonce pour modifier sa catégorie
- Créer un endpoint GET pour lire tous les catégories
  - Permettre de les filtrer par nom pour un champ d'autocomplete
- Modifier le endpoint GET des annonces pour les filtrer par catégorie

Atelier : Tag - ManyToMany

- Créer l'entité Tag
- Renseigner l'entité dans la datasource
- Modifier le endpoint POST d'une annonce pour ajouter plusieurs tags
- Modifier le endpoint PUT d'une annonce modifier les tags d'une annonce
- Créer un endpoint GET pour lire tous les tags
  - Permettre de les filtrer par nom pour un champ d'autocomplete
- Créer un endpoint DELETE pour supprimer un tag
- Modifier le endpoint GET des annonces pour les filtrer par tags