Persistance de données avec **TypeORM**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

TYPEORM

Plan

Plan

Plan

Plan

Plan

TypeORM

Limites du package **mysql** et **mysql2**

- Nécessité de formuler des requêtes **SQL**
- Logiques relationnelle et objet dans un même fichier
- Le développeur doit tout écrire et gérer : ouverture et fermeture de connexion, gestion de transaction, préparation des requêtes pour faire le **CRUD** : (`Create`, `Read`, `Update`, **et** `Delete`)...

TypeORM

Limites du package **mysql** et **mysql2**

- Nécessité de formuler des requêtes **SQL**
- Logiques relationnelle et objet dans un même fichier
- Le développeur doit tout écrire et gérer : ouverture et fermeture de connexion, gestion de transaction, préparation des requêtes pour faire le **CRUD** : (`Create`, `Read`, `Update`, et `Delete`)...

Une meilleure solution ?

Utiliser un **ORM**...

TypeORM

ORM : **O**bject-**R**elational **M**apping (lien objet-relationnel)

- Couche d'abstraction à la base de données
- Permettant au développeur d'utiliser les tables d'une base de données comme des objets
- Consistant à associer :
  - une ou plusieurs classes à chaque table
  - un attribut de classe à chaque colonne de la table

TypeORM

ORM : avantages

- Plus besoin d'écrire de requête **SQL**
- Suppression d'une très grande partie du code nécessaire avec **mysql** et **mysql2**
  - Gain de temps
  - Simplification du code source
- Code portable indépendant vis-à-vis de **SGBD**

TypeORM

ORM : avantages

- Plus besoin d'écrire de requête **SQL**
- Suppression d'une très grande partie du code nécessaire avec **mysql** et **mysql2**
  - Gain de temps
  - Simplification du code source
- Code portable indépendant vis-à-vis de **SGBD**

ORM : inconvénients

- Complexité (d'apprentissage)
- Problème d'optimalité pour les requêtes complexes modifiant plusieurs tuples

TypeORM

Autres frameworks **ORM** pour **Node.js**

- **TypeORM**
- Sequelize
- Prisma
- Objection.js
- Mongoose
- ...

TypeORM

TypeORM

- Framework ORM pour **Node.js**
- Open-source et écrit en **TypeScript**
- Inspiré par `Hibernate` (**Java**), `Doctrine` (**PHP**) et `EntityFramework` (**Microsoft .NET**)
- Pouvant être utilisé avec **TypeScript** et **JavaScript** (ES5, ES6, ES7, ES8)
- Supportant plusieurs SGBD : **MySQL**, **PostgreSQL**, **MariaDB**, **SQLite**, **SQL Server**, **Oracle**...

TypeORM

Quel **E**nvironnement de **D**éveloppement **I**ntégré (IDE) pour **Vue.js** ?

- **Visual Studio Code** (À ne pas confondre avec Visual Studio)
- Eclipse
- ...

TypeORM

Quel **E**nvironnement de **D**éveloppement **I**ntégré (IDE) pour **Vue.js** ?

- **Visual Studio Code** (À ne pas confondre avec Visual Studio)
- Eclipse
- ...

Visual **S**tudio **C**ode (ou **VSC**), pourquoi ?

- Gratuit.
- Extensible selon le langage de programmation.
- Recommandé par les développeurs Front-end.

TypeORM

VSC : téléchargement

`code.visualstudio.com/download`

TypeORM

Quelques raccourcis **VSC**

- Pour activer la sauvegarde automatique : aller dans `File > AutoSave`
- Pour indenter son code : `Alt` `Shift` `f`
- Pour commenter/décommenter : `Ctrl` `:`
- Pour sélectionner toutes les occurrences : `Ctrl` `f2`
- Pour sélectionner l'occurrence suivante : `Ctrl` `d`
- Pour placer le curseur dans plusieurs endroits différents : `Alt`

TypeORM

Commençons par installer `TypeORM`

```
npm install typeorm --save
```

TypeORM

Commençons par installer `TypeORM`

```
npm install typeorm --save
```

Ensuite `reflect-metadata`

```
npm install reflect-metadata --save
```

TypeORM

Commençons par installer `TypeORM`

```
npm install typeorm --save
```

Ensuite `reflect-metadata`

```
npm install reflect-metadata --save
```

Et enfin le package de typage pour `Node.js`

```
npm install @types/node --save-dev
```

TypeORM

Avant de commencer, installer **MySQL** (s'il n'est pas installé)

- Aller à `https://dev.mysql.com/downloads/mysql/` et choisir la version à télécharger selon le système d'exploitation
- Lancer l'installation du fichier MSI sous Windows (fichier pkg de l'archive DMG sous MAC)

TypeORM

Pour installer le driver de MySQL

```
npm install mysql2 --save
```

TypeORM

Pour générer un projet `TypeORM`

```
npx typeorm init --name cours-typeorm --database mysql
```

TypeORM

Pour générer un projet `TypeORM`

```
npx typeorm init --name cours-typeorm --database mysql
```

Explication

- `--name cours-typeorm` : nom du projet
- `--database mysql` : nom du SGBD

TypeORM

Structure du projet généré (Documentation officielle)

```
MyProject
├── src                    // place of your TypeScript code
│   ├── entity             // place where your entities (database models) are stored
│   │   └── User.ts        // sample entity
│   ├── migration          // place where your migrations are stored
│   ├── data-source.ts     // data source and all connection configuration
│   └── index.ts           // start point of your application
├── .gitignore             // standard gitignore file
├── package.json           // node module dependencies
├── README.md              // simple readme file
└── tsconfig.json          // TypeScript compiler options
```

TypeORM

Installons ensuite `mysql2`

```
npm install mysql2 --save
```

TypeORM

Installons ensuite `mysql2`

```
npm install mysql2 --save
```

Désinstallons `mysql`

```
npm uninstall mysql --save
```

TypeORM

Allons vérifier le contenu suivant dans `package.json`

```
{
   "name": "cours-typeorm",
   "version": "0.0.1",
   "description": "Awesome project developed with TypeORM.",
   "type": "commonjs",
   "devDependencies": {
      "@types/node": "^16.11.10",
      "ts-node": "10.7.0",
      "typescript": "4.5.2"
   },
   "dependencies": {
      "mysql2": "^3.2.0",
      "reflect-metadata": "^0.1.13",
      "typeorm": "0.3.12"
   },
   "scripts": {
      "start": "ts-node src/index.ts",
      "typeorm": "typeorm-ts-node-commonjs"
   }
}
```

TypeORM

Modifions le contenu de `src/data-source.ts` en fonction de nos données de connexion

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User],
    migrations: [],
    subscribers: [],
})
```

TypeORM

Modifions le contenu de `src/data-source.ts` en fonction de nos données de connexion

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User],
    migrations: [],
    subscribers: [],
})
```

N'oublions pas de créer la base de données `cours_typeorm`.

TypeORM

Pour lancer le projet

```
npm start
```

TypeORM

Entité

- Représentant d'une (ou plusieurs) table·s d'une base de données relationnelle dans le modèle objet.
- Classe Persistante : Classes + méta-données indispensables pour le mapping
- Pouvant être une classe concrète ou abstraite
- Pouvant étendre des classes non-entités ou des classes entités

TypeORM

Mapping

- Une entité $\simeq$ une table
- Un attribut simple $\simeq$ une colonne
- Un attribut objet ou une collection $\simeq$ une table + une clé étrangère
- Une clé primaire avec méta-donné `id` $\simeq$ clé primaire

TypeORM

typeORM **L'entité générée par**

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number

}
```

TypeORM

Explication

- `@Entity()` : déclare la classe comme entité
- `@Column()` : déclare l'attribut comme colonne
- `@PrimaryGeneratedColumn()` : déclare l'attribut comme clé primaire avec une valeur auto-incrémentale

TypeORM

Contenu d'`index.ts`

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    console.log("Inserting a new user into the database...")
    const user = new User()
    user.firstName = "Timber"
    user.lastName = "Saw"
    user.age = 25
    await AppDataSource.manager.save(user)
    console.log("Saved a new user with id: " + user.id)

    console.log("Loading users from the database...")
    const users = await AppDataSource.manager.find(User)
    console.log("Loaded users: ", users)

    console.log("Here you can setup and run express / fastify / any other framework.")

}).catch(error => console.log(error))
```

TypeORM

Pour chercher une personne selon son identifiant

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const u = await AppDataSource.manager.findOneBy(User, { id: 1 });
    console.log(u)

}).catch(error => console.log(error))
```

TypeORM

On peut aussi chercher selon plusieurs critères

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const u = await AppDataSource.manager.findOneBy(User, {
        firstName: "Timber",
        lastName: 'Saw'
    });
    console.log(u)

}).catch(error => console.log(error))
```

TypeORM

Pour sélectionner tous les tuples qui correspondent aux critères de recherche

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const users = await AppDataSource.manager.findBy(User, {
        firstName: "Timber",
        lastName: 'Saw'
    });
    console.log(users)

}).catch(error => console.log(error))
```

TypeORM

Pour sélectionner et compter les données d'une table

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const [users, nombre] = await AppDataSource.manager.findAndCount(
       User);
    console.log(users, nombre)

}).catch(error => console.log(error))
```

TypeORM

Pour sélectionner et compter les données d'une table respectant une ou plusieurs conditions

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const [users, nombre] = await AppDataSource.manager.findAndCountBy(
       User,
         {
             lastName: 'Saw'
         }
    )

    console.log(users, nombre)

}).catch(error => console.log(error))
```

TypeORM

Pour compter tous les tuples d'une table

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const  nombre = await AppDataSource.manager.count(User)

    console.log(nombre)

}).catch(error => console.log(error))
```

TypeORM

Pour compter tous les tuples d'une table respectant une ou plusieurs conditions

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const nombre = await AppDataSource.manager.countBy(User,
        {
            lastName: 'Saw'
        }
    )

    console.log(nombre)

}).catch(error => console.log(error))
```

TypeORM

Pour modifier une personne

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const u = await AppDataSource.manager.findOneBy(User, { id: 1 });
    u.firstName = 'John'
    u.lastName = 'Doe'
    AppDataSource.manager.save(u)
        .then(() => console.log("Modification effectuée aves succès"))
        .catch((err) => console.log(err))

}).catch(error => console.log(error))
```

TypeORM

Pour supprimer un utilisateur

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const u = await AppDataSource.manager.findOneBy(User, { id: 1 });
    AppDataSource.manager.remove(u)
        .then(() => console.log("Suppression effectuée aves succès"))
        .catch((err) => console.log(err))

}).catch(error => console.log(error))
```

TypeORM

Remarque

Pour le **CRUD**, **TypeORM** nous permet aussi d'utiliser un **Repository** (un pour chaque entité)

TypeORM

Pour récupérer le Repository de l'entité `User`

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);

}).catch(error => console.log(error))
```

TypeORM

Pour chercher un utilisateur selon son identifiant, plus besoin de spécifier son entité (`User`)

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const u = await userRepository.findOneBy({ id: 2 });

    console.log(u);

}).catch(error => console.log(error))
```

TypeORM

Options

- La méthode de recherche `find...By` accepte un paramètre `options`
- Le paramètre `options` permet de spécifier les critères de recherche

TypeORM

Exemple avec `Not`

```
import { Not } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        firstName: Not("Dalton")
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Exemple avec `Not`

```
import { Not } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        firstName: Not("Dalton")
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE firstName != 'Dalton'
```

TypeORM

Exemple avec `Like`

```
import { Like } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        firstName: Like("%a%")
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Exemple avec `Like`

```
import { Like } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        firstName: Like("%a%")
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE firstName LIKE '%a%'
```

TypeORM

Exemple avec `Between`

```
import { Between } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: Between(30, 50)
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Exemple avec `Between`

```
import { Between } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: Between(30, 50)
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE age BETWEEN 30 AND 50
```

TypeORM

Exemple avec `In`

```
import { In } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: In([30, 50])
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Exemple avec `In`

```
import { In } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: In([30, 50])
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE age IN (30, 50)
```

TypeORM

On peut aussi imbriquer les options (exemple avec `Not` et `In`)

```
import { In, Not } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: Not(In([30, 50]))
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

On peut aussi imbriquer les options (exemple avec `Not` et `In`)

```
import { In, Not } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.findBy({
        age: Not(In([30, 50]))
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE age NOT IN (30, 50)
```

TypeORM

Autres opérateurs

- `LessThan`
- `LessThanOrEqual`
- `MoreThan`
- `MoreThanOrEqual`
- `Equal`
- `ILike`
- `Any`
- `IsNull`

TypeORM

Options

- Les autres méthodes de recherche acceptent un paramètre `options` contenant les clauses **SQL** comme (`select`, `where`, `order by`...)
- Le paramètre `options` permet de
  - spécifier les colonnes à sélectionner
  - spécifier les critères de recherche
  - ordonner le résultat
  - ...

TypeORM

Pour sélectionner uniquement `firstName` **et** `lastName`

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        select: {
            firstName: true,
            lastName: true
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour sélectionner uniquement `firstName` et `lastName`

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        select: {
            firstName: true,
            lastName: true
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT firstName, lastName FROM user
```

TypeORM

Pour ajouter une condition

```
import { MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        select: {
            firstName: true,
            lastName: true
        },
        where: {
            age: MoreThan(30)
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour ajouter une condition

```
import { MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        select: {
            firstName: true,
            lastName: true
        },
        where: {
            age: MoreThan(30)
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT firstName, lastName FROM user WHERE age > 30
```

TypeORM

Exemple avec plusieurs conditions

```
import { Like, MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        where: {
            age: MoreThan(30),
            firstName: Like('%a%')
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Exemple avec plusieurs conditions

```
import { Like, MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        where: {
            age: MoreThan(30),
            firstName: Like('%a%')
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT firstName, lastName FROM user WHERE age > 30 AND firstName LIKE '%a%'
```

TypeORM

Pour enchaîner les conditions avec `OR`

```
import { Like, MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        where: [
            { age: MoreThan(30) },
            { firstName: Like('%a%') }
        ]
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour enchaîner les conditions avec `OR`

```
import { Like, MoreThan } from "typeorm";
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        where: [
            { age: MoreThan(30) },
            { firstName: Like('%a%') }
        ]
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT firstName, lastName FROM user WHERE age > 30 OR firstName LIKE '%a%'
```

TypeORM

Pour ordonner le résultat

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        order: {
            firstName: 'ASC',
            lastName: 'DESC'
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour ordonner le résultat

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        order: {
            firstName: 'ASC',
            lastName: 'DESC'
        }
    });

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user
ORDER BY firstName ASC, lastName DESC
```

TypeORM

Pour limiter le nombre de tuples sélectionnés

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        take: 2
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour limiter le nombre de tuples sélectionnés

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        take: 2
    });

    console.log(users);

}).catch(error => console.log(error))
```


Requête SQL exécutée

```
SELECT * FROM user
LIMIT 2
```

TypeORM

Pour limiter le nombre de tuples sélectionnés

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        skip: 1,
        take: 2
    });

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour limiter le nombre de tuples sélectionnés

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository.find({
        skip: 1,
        take: 2
    });

    console.log(users);

}).catch(error => console.log(error))
```


Requête SQL exécutée

```
SELECT * FROM user
LIMIT 2
OFFSET 1
```

TypeORM

Remarque

Pour **MySQL** et **MicroSoft SQL Server**, `skip` doit être toujours accompagné par `take`.

TypeORM

Autres annotations

Décorateur	désignation
`@Entity`	permet de qualifier la classe comme entité
`@PrimaryColumn`	indique l'attribut qui correspond à la clé primaire de la table
`@PrimaryColumnGenerated`	permet la génération de valeurs pour la clé primaire
`@Column`	définit les propriétés d'une colonne
`@IdClass`	indique que l'entité contient une clé composée
`@Generated`	permet la génération de valeurs pour n'importe quelle colonne

TypeORM

Attributs de l'annotation `@Column`

Attribut	**désignation**
`name`	définit un nom de colonne différent de celui de l'attribut
`length`	permet de fixer la longueur d'une chaîne de caractères
`unique`	indique que la valeur d'un champ est unique
`nullable`	précise si un champ est null (ou non)
`default`	précise la valeur par défaut de ce champ
...	...

TypeORM

Quatre (ou trois) relations possibles

- `OneToOne` : chaque objet d'une première classe est en relation avec un seul objet de la deuxième classe
- `OneToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe (la réciproque est `ManyToOne`)
- `ManyToMany` : chaque objet d'une première classe peut être en relation avec plusieurs objets de la deuxième classe et inversement

TypeORM

Commençons par créer une entité `Account` dans `entity`

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class Account {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    userName: string

    @Column()
    password: string

}
```

TypeORM

Sans oublier de déclarer `Account` dans `data-source.ts`

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { Account } from "./entity/Account"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User, Account],
    migrations: [],
    subscribers: [],
})
```

TypeORM

Déclarons `Account` **dans** `User`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne } from "typeorm"
import { Account } from './Account'

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number

    @OneToOne(() => Account)
    @JoinColumn()
    account: Account
}
```

TypeORM

Appellation

- `Personne` : entité propriétaire
- `Adresse` : entité inverse

TypeORM

Pour persister un `User` avec un `Account`

```
import { AppDataSource } from "./data-source"
import { Account } from "./entity/Account";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const accountRepository = AppDataSource.getRepository(Account);
    const account: Account = new Account()
    account.userName = 'doe'
    account.password = 'doe'
    accountRepository.save(account)
    const user = new User()
    user.firstName = "Timber"
    user.lastName = "Saw"
    user.age = 25
    user.account = account
    userRepository.save(user)
    console.log(user);

}).catch(error => console.log(error))
```

TypeORM

Appellation

- `Personne` : entité propriétaire
- `Adresse` : entité inverse

TypeORM

Pour persister `account` et `user` avec un seul `save`, on ajoute l'attribut `cascade` dans le décorateur `@OneToOne`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne } from "typeorm"
import { Account } from './Account'

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number

    @OneToOne(() => Account, { cascade: ['insert'] })
    @JoinColumn()
    account: Account

}
```

TypeORM

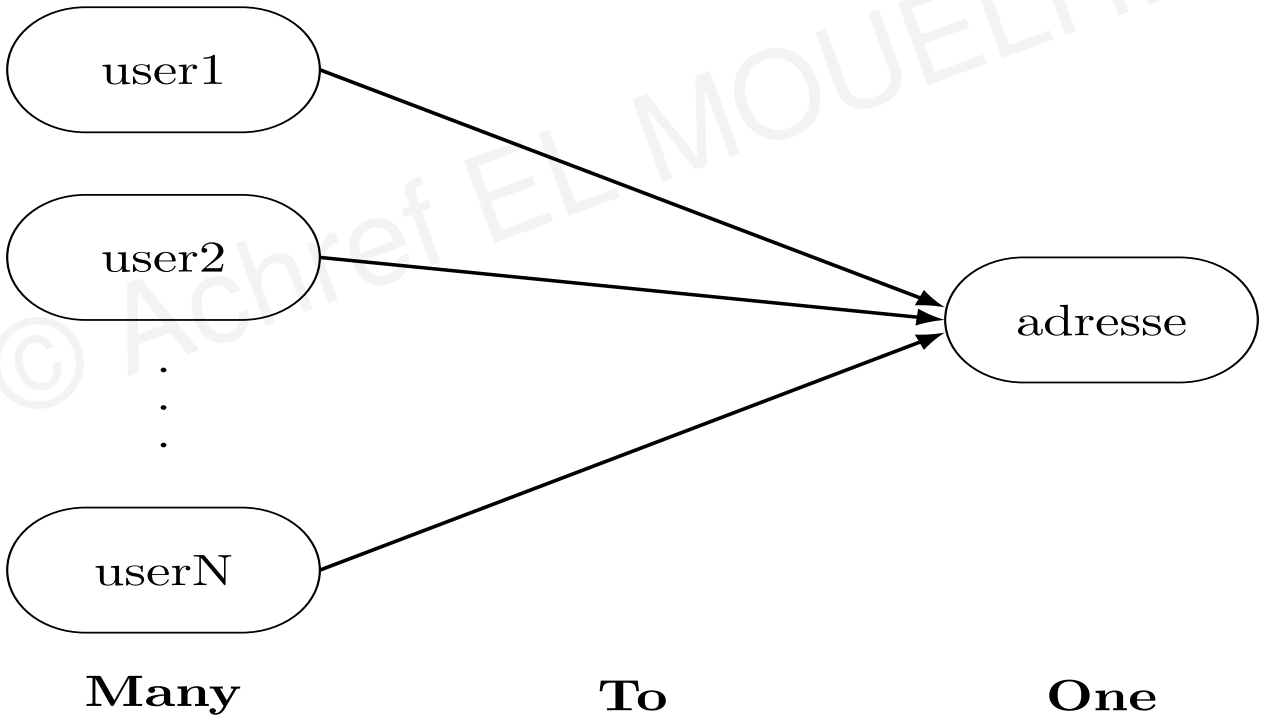
Testons avec un seul `persist`

```
import { AppDataSource } from "./data-source"
import { Account } from "./entity/Account";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const account: Account = new Account()
    account.userName = 'doe'
    account.password = 'doe'
    const user = new User()
    user.firstName = "Timber"
    user.lastName = "Saw"
    user.age = 25
    user.account = account
    userRepository.save(user)
    console.log(user);

}).catch(error => console.log(error))
```

TypeORM

Hypothèse

Si plusieurs personnes pouvaient avoir la même adresse.



TypeORM

Il faut juste remplacer `@ManyToOne` **par** `@OneToOne`

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"

@Entity()
export class Adresse {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    rue: string

    @Column()
    codePostal: string

    @Column()
    ville: string
}
```

TypeORM

Sans oublier de déclarer `Account` dans `data-source.ts`

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { Account } from "./entity/Account"
import { Adresse } from "./entity/Adresse"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User, Account, Adresse],
    migrations: [],
    subscribers: [],
})
```

TypeORM

Déclarons `Adresse` **dans** `User`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne } from "
   typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"

@Entity()
export class User {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    firstName: string

    @Column()
    lastName: string

    @Column()
    age: number

    @OneToOne(() => Account, { cascade: ['insert'] })
    @JoinColumn()
    account: Account

    @ManyToOne(() => User, { cascade: ['insert'] })
    @JoinColumn()
    adresse: Adresse
}
```

TypeORM

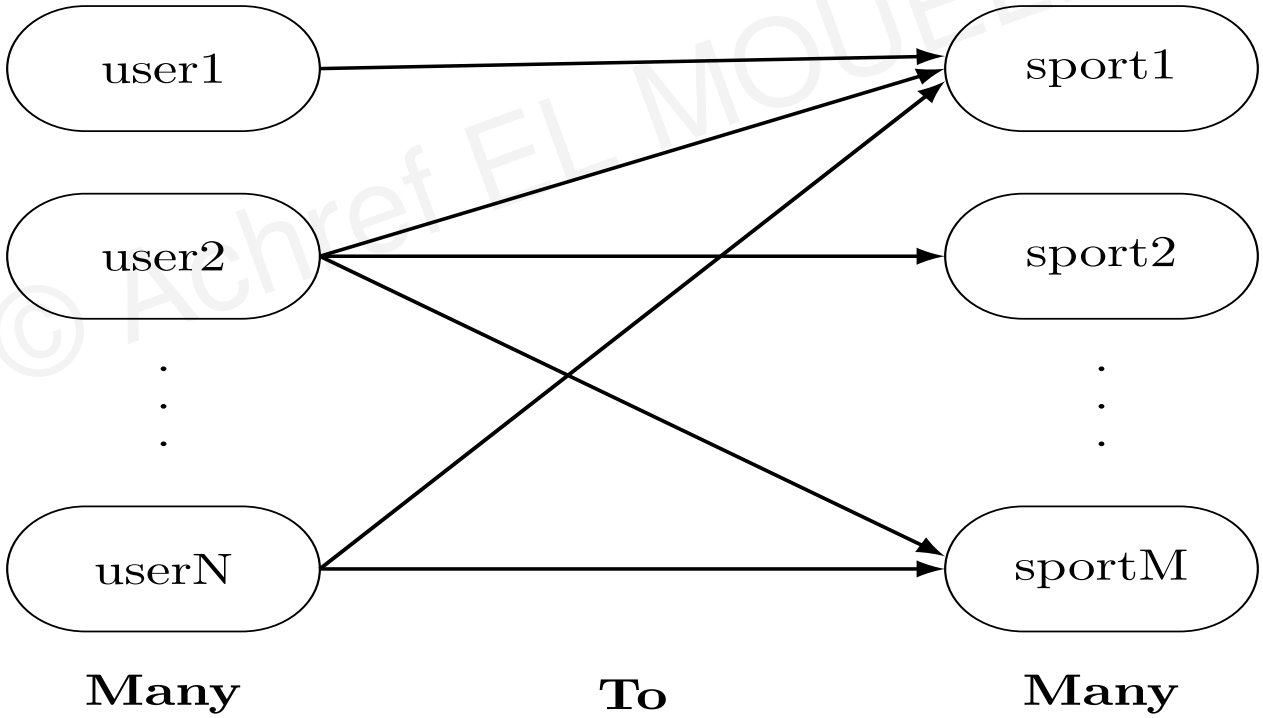
Et pour tester

```
import { AppDataSource } from "./data-source"
import { Adresse } from "./entity/Adresse";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const adresse = new Adresse()
    adresse.rue = 'paradis'
    adresse.codePostal = '13006'
    adresse.ville = 'Marseille'
    const user1 = new User()
    user1.firstName = "Timber"
    user1.lastName = "Saw"
    user1.age = 25
    user1.adresse = adresse
    const user2 = new User()
    user2.firstName = "Dalton"
    user2.lastName = "Jack"
    user2.age = 45
    user2.adresse = adresse
    userRepository.save(user1)
    userRepository.save(user2)
    console.log(user1, user2);

}).catch(error => console.log(error))
```

TypeORM

Exemple

- Un utilisateur peut pratiquer 0, 1 ou plusieurs sports
- Un sport peut être pratiqué par 0, 1 ou plusieurs utilisateurs



TypeORM

Démarche

- Créer une énumération `SportType`
- Créer une entité `Sport` qui utilise `SportType`
- Définir la relation `ManyToMany` (exactement comme les deux relations précédentes) soit dans `User` soit dans `Sport`

TypeORM

Commençons par créer `SportType` dans un répertoire `enum`

```
export enum SportType {
    INDIVIDUEL = "ind",
    COLLECTIF = "col",
}
```

TypeORM

Ensuite l'entité `Sport`

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm"
import { SportType } from "../../enum/SportType"

@Entity()
export class Sport {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    nom: string

    @Column({
        type: "enum",
        enum: SportType,
        default: SportType.INDIVIDUEL,
    })
    type: SportType
}
```

TypeORM

Sans oublier de déclarer `Sport` **dans** `data-source.ts`

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { Account } from "./entity/Account"
import { Adresse } from "./entity/Adresse"
import { Sport } from "./entity/Sport"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User, Account, Adresse, Sport],
    migrations: [],
    subscribers: [],
})
```

TypeORM

Ajoutons le `ManyToMany` dans la classe `User` : `@JoinTable()` est indispensable pour la génération de la table de jointure

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne, ManyToMany,
   JoinTable } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
export class User {

    // les attributs précédents

    @ManyToMany(() => Sport, { cascade: ['insert'] })
    @JoinTable()
    sports: Sport[]

}
```

Pour tester

```
import { SportType } from "../enum/SportType";
import { AppDataSource } from "./data-source"
import { Adresse } from "./entity/Adresse";
import { Sport } from "./entity/Sport";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const sport1 = new Sport()
    sport1.nom = "foot"
    sport1.type = SportType.COLLECTIF
    const sport2 = new Sport()
    sport2.nom = "tennis"
    sport2.type = SportType.INDIVIDUEL
    const user1 = new User()
    user1.firstName = "Timber"
    user1.lastName = "Saw"
    user1.age = 25
    user1.sports = [sport1, sport2]
    userRepository.save(user1)
    const user2 = new User()
    user2.firstName = "Dalton"
    user2.lastName = "Jack"
    user2.age = 45
    user2.sports = [sport1]
    userRepository.save(user2)
    console.log(user1, user2);

}).catch(error => console.log(error))
```

TypeORM

Constats

- Deux tables créées (sans compter les précédentes) :
  - `sport` avec les colonnes `id, nom, type`
  - `user_sports_sport` avec les colonnes `#userId, #sportId`

TypeORM

En récupérant un utilisateur depuis la base de données, les sports n'apparaissent pas dans le résultat

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOneBy({ id: 1 })
    console.log(user);

}).catch(error => console.log(error))
```

TypeORM

Explication

- Par défaut, **TypeORM** ne charge pas les entités inverses
- **TypeORM** propose deux modes de chargement pour les entités inverses
  - `eager` : charger les entités inverses avec l'entité propriétaire
  - `lazy` : charger les entités inverses sur demande (avec les promesses)

TypeORM

Ajoutons la propriété `eager` dans le décorateur `@ManyToMany`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne, ManyToMany,
   JoinTable } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
export class User {

    // les autres attributs

    @ManyToMany(() => Sport,
        {
            cascade: ['insert'],
            eager: true,
        })
    @JoinTable()
    sports: Sport[]
}
```

TypeORM

Testons le code précédent et vérifions que les sports sont présents dans le résultat

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOneBy({ id: 1 })
    console.log(user);

}).catch(error => console.log(error))
```

TypeORM

Pour le lazy loading, on utilise les promesses

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne, ManyToMany,
   JoinTable } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
export class User {

    // les autres attributs

    @ManyToMany(() => Sport, //(sport) => sport.users,
        {
            cascade: ['insert'],
        })
    @JoinTable()
    sports: Promise<Sport[]>
}
```

TypeORM

Testons le code précédent et vérifions que les sports ne sont pas présents dans le résultat

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOneBy({ id: 1 })
    console.log(user);

}).catch(error => console.log(error))
```

TypeORM

Les sports sont chargés sur demande

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOneBy({ id: 1 })
    console.log(await user.sports);

}).catch(error => console.log(error))
```

TypeORM

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

TypeORM

Si la classe association est porteuse de données

- Par exemple : la relation (ArticleCommande) entre Commande et Article
- Il faut préciser la quantité de chaque article dans une commande

Solution

- Créer trois entités `Article`, `Commande` et `ArticleCommande`
- Définir la relation `OneToMany` entre `Article` et `ArticleCommande`
- Définir la relation `ManyToOne` entre `ArticleCommande` et `Commande`

TypeORM

Remarques

- Les associations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `personne.sports`
- Mais on ne peut faire `sport.personnes`

TypeORM

Remarques

- Les associations, qu'on a étudiées, sont unidirectionnelles
- C'est à dire on peut faire `personne.sports`
- Mais on ne peut faire `sport.personnes`

Solution

Rendre les relations bidirectionnelles

TypeORM

Modifier l'entité inverse `Sport`

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany } from "typeorm"
import { SportType } from "../../enum/SportType"
import { User } from "./User"

@Entity()
export class Sport {

    @PrimaryGeneratedColumn()
    id: number

    @Column()
    nom: string

    @Column({
        type: "enum",
        enum: SportType,
        default: SportType.INDIVIDUEL,
    })
    type: SportType

    @ManyToMany(() => User, (user) => user.sports, {
        eager: true,
    })
    users: Promise<User[]>

}
```

TypeORM

Nouveau contenu de `User`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne, ManyToMany,
   JoinTable } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
export class User {

    // les attributs précédents

    @ManyToMany(() => Sport, (sport) => sport.users,
        {
            cascade: ['insert'],
        })
    @JoinTable()
    sports: Promise<Sport[]>
}
```

TypeORM

Ainsi, on peut faire

```
import { AppDataSource } from "./data-source"
import { Sport } from "./entity/Sport";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOneBy({ id: 1 })
    console.log(await user.sports);
    const sportRepository = AppDataSource.getRepository(Sport);
    const sport = await sportRepository.findOneBy({ id: 1 })
    console.log(await sport.users)

}).catch(error => console.log(error))
```

TypeORM

Et pour tester (`save`)

```
import { SportType } from "../enum/SportType";
import { AppDataSource } from "./data-source"
import { Sport } from "./entity/Sport";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const sport1 = new Sport()
    sport1.nom = "foot"
    sport1.type = SportType.COLLECTIF
    const sport2 = new Sport()
    sport2.nom = "tennis"
    sport2.type = SportType.INDIVIDUEL
    const user1 = new User()
    user1.firstName = "Timber"
    user1.lastName = "Saw"
    user1.age = 25
    user1.sports = Promise.resolve([sport1, sport2])
    userRepository.save(user1)
    const user2 = new User()
    user2.firstName = "Dalton"
    user2.lastName = "Jack"
    user2.age = 45
    user2.sports = Promise.resolve([sport1])
    userRepository.save(user2)
    console.log(user1, user2);

}).catch(error => console.log(error))
```

TypeORM

Pour définir une relation bidirectionnelle entre deux entités

- si la relation définie dans l'entité propriétaire est `OneToMany`, alors dans l'entité inverse la relation sera `ManyToOne`, et inversement.
- si la relation définie dans l'entité propriétaire est `OneToOne`, alors dans l'entité inverse la relation sera aussi `OneToOne`.
- si la relation définie dans l'entité propriétaire est `ManyToMany`, alors dans l'entité inverse la relation sera aussi `ManyToMany`.

TypeORM

Trois solutions pour la transformation de l'héritage

- `SINGLE_TABLE`
- `CONCRETE_TABLE`

TypeORM

Exemple

- Une classe mère `User`
- Deux classes filles `Etudiant` et `Enseignant`

TypeORM

Exemple

- Une classe mère `User`
- Deux classes filles `Etudiant` et `Enseignant`

La classe `Etudiant`

```
import { Column } from "typeorm";
import { User } from "./User";

export class Etudiant extends User {
    @Column()
    niveau: string
}
```

La classe `Enseignant`

```
import { Column } from "typeorm";
import { User } from "./User";

export class Enseignant extends User {
    @Column()
    salaire: number
}
```

TypeORM

Pour utiliser la solution SINGLE_TABLE

- On décore la super-classe (ici `User`) par `@TableInheritance({ column: { type: "varchar", name: "type" } })` sur la classe `User`.
- Les sous-classes doivent être décorées par `@ChildEntity()`

TypeORM

Pour utiliser la solution SINGLE_TABLE

- On décore la super-classe (ici `User`) par `@TableInheritance({ column: { type: "varchar", name: "type" } })` sur la classe `User`.
- Les sous-classes doivent être décorées par `@ChildEntity()`

Explication

Pour distinguer Étudiant, Enseignant, et User, **TypeORM** stockera dans la colonne `type` respectivement les valeurs suivantes

- `Etudiant`
- `Enseignant`
- `User`

TypeORM

La classe `User`

```
import { Entity, PrimaryGeneratedColumn, Column, JoinColumn, OneToOne, ManyToOne, ManyToMany,
   JoinTable, TableInheritance } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
@TableInheritance({ column: { type: "varchar", name: "type" } })
export class User {

  // + tout le code précédent

}
```


La classe `Etudiant`

```
import { ChildEntity, Column } from "typeorm";
import { User } from "./User";

@ChildEntity()
export class Etudiant extends User {

    @Column()
    niveau: string
}
```

La classe `Enseignant`

```
import { ChildEntity, Column } from "typeorm";
import { User } from "./User";

@ChildEntity()
export class Enseignant extends User {

    @Column()
    salaire: number
}
```

TypeORM

Sans oublier de déclarer `Etudiant` **et** `Enseignant` **dans** `data-source.ts`

```
import "reflect-metadata"
import { DataSource } from "typeorm"
import { Account } from "./entity/Account"
import { Adresse } from "./entity/Adresse"
import { Enseignant } from "./entity/Enseignant"
import { Etudiant } from "./entity/Etudiant"
import { Sport } from "./entity/Sport"
import { User } from "./entity/User"

export const AppDataSource = new DataSource({
    type: "mysql",
    host: "localhost",
    port: 3306,
    username: "root",
    password: "",
    database: "cours_typeorm",
    synchronize: true,
    logging: false,
    entities: [User, Account, Adresse, Sport, Etudiant, Enseignant],
    migrations: [],
    subscribers: [],
})
```

Et pour tester

```
import { SportType } from "../enum/SportType";
import { AppDataSource } from "./data-source"
import { Enseignant } from "./entity/Enseignant";
import { Etudiant } from "./entity/Etudiant";
import { Sport } from "./entity/Sport";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const etudiantRepository = AppDataSource.getRepository(Etudiant);
    const enseignantRepository = AppDataSource.getRepository(Enseignant);
    const user = new User()
    user.firstName = "Dalton"
    user.lastName = "Jack"
    user.age = 45
    userRepository.save(user)
    const etudiant = new Etudiant()
    etudiant.firstName = "Maggio"
    etudiant.lastName = "Candice"
    etudiant.age = 20
    etudiant.niveau = 'Doctorat'
    etudiantRepository.save(etudiant)
    const enseignant = new Enseignant()
    enseignant.firstName = "Baggio"
    enseignant.lastName = "Roberto"
    enseignant.age = 55
    enseignant.salaire = 1700
    enseignantRepository.save(enseignant)
    console.log(user, etudiant, enseignant);

}).catch(error => console.log(error))
```

TypeORM

Constats

- Trois colonnes ajoutées à la table `personne` :
  - `TYPR_PERSONNE`
  - `niveau`
  - `salaire`
- Trois tuples ajoutés

TypeORM

CONCRETE_TABLE

- Supprimer le décorateur `@TableInheritance` dans `User.ts`
- Remplacer `ChildEntity` par `Entity`

TypeORM

CONCRETE_TABLE

- Supprimer le décorateur `@TableInheritance` dans `User.ts`
- Remplacer `ChildEntity` par `Entity`

La classe `Etudiant`

```
import { Column, Entity } from "typeorm";
import { User } from "./User";

@Entity()
export class Etudiant extends User {
    @Column()
    niveau: string
}
```

La classe `Enseignant`

```
import { Column, Entity } from "typeorm";
import { User } from "./User";

@Entity()
export class Enseignant extends User {
    @Column()
    salaire: number
}
```

TypeORM

Nouveau contenu de `User`

```
import { PrimaryGeneratedColumn, Column, JoinColumn, OneToOne,
   ManyToOne, ManyToMany, JoinTable, Entity } from "typeorm"
import { Account } from './Account'
import { Adresse } from "./Adresse"
import { Sport } from "./Sport"

@Entity()
export abstract class User {

  //  le code précédent

}
```

TypeORM

Et pour tester

```
import { SportType } from "../enum/SportType";
import { AppDataSource } from "./data-source"
import { Enseignant } from "./entity/Enseignant";
import { Etudiant } from "./entity/Etudiant";
import { Sport } from "./entity/Sport";
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const etudiantRepository = AppDataSource.getRepository(Etudiant);
    const enseignantRepository = AppDataSource.getRepository(Enseignant);

    const etudiant = new Etudiant()
    etudiant.firstName = "Maggio"
    etudiant.lastName = "Candice"
    etudiant.age = 20
    etudiant.niveau = 'Doctorat'
    etudiantRepository.save(etudiant)
    const enseignant = new Enseignant()
    enseignant.firstName = "Baggio"
    enseignant.lastName = "Roberto"
    enseignant.age = 55
    enseignant.salaire = 1700
    enseignantRepository.save(enseignant)
    console.log(etudiant, enseignant);

}).catch(error => console.log(error))
```

TypeORM

Query Builder

- Constructeur de requête **SQL** en mode objet
- Facilite le mapping de données
- Permet d'exécuter des requêtes de type
  - `SELECT`
  - `INSERT`
  - `UPDATE`
  - ...

TypeORM

Pour sélectionner un seul utilisateur selon son identifiant

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const firstUser = await userRepository
        .createQueryBuilder("user")
        .where("user.id = :id", { id: 1 })
        .getOne()

    console.log(firstUser);

}).catch(error => console.log(error))
```

TypeORM

Pour sélectionner un seul utilisateur selon son identifiant

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const firstUser = await userRepository
        .createQueryBuilder("user")
        .where("user.id = :id", { id: 1 })
        .getOne()

    console.log(firstUser);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE id = 1
```

TypeORM

Pour lister plusieurs conditions

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const firstUser = await userRepository
        .createQueryBuilder("user")
        .where("user.firstName = :firstName", { firstName: "Dalton" })
        .andWhere("user.lastName = :lastName", { lastName: "Jack" })
        .getOne()

    console.log(firstUser);

}).catch(error => console.log(error))
```

TypeORM

Pour lister plusieurs conditions

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const firstUser = await userRepository
        .createQueryBuilder("user")
        .where("user.firstName = :firstName", { firstName: "Dalton" })
        .andWhere("user.lastName = :lastName", { lastName: "Jack" })
        .getOne()

    console.log(firstUser);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE firstName = 'Dalton' AND lastName = 'Jack'
```

TypeORM

Pour récupérer tous les résultats

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository
        .createQueryBuilder("user")
        .where("user.firstName = :firstName", { firstName: "Dalton" })
        .andWhere("user.lastName = :lastName", { lastName: "Jack" })
        .getMany()

    console.log(users);

}).catch(error => console.log(error))
```

TypeORM

Pour récupérer tous les résultats

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const users = await userRepository
        .createQueryBuilder("user")
        .where("user.firstName = :firstName", { firstName: "Dalton" })
        .andWhere("user.lastName = :lastName", { lastName: "Jack" })
        .getMany()

    console.log(users);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
SELECT * FROM user WHERE firstName = 'Dalton' AND lastName = 'Jack'
```

TypeORM

Pour insérer des nouveaux tuples dans la base de données

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .insert()
        .into(User)
        .values([
            { firstName: "Hotchner", lastName: "Maria" },
            { firstName: "Phantom", lastName: "Lancer" },
        ])
        .execute()
        console.log(`Nombre de tuples insérés : ${count.raw.affectedRows}`);

}).catch(error => console.log(error))
```

TypeORM

Pour insérer des nouveaux tuples dans la base de données

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .insert()
        .into(User)
        .values([
            { firstName: "Hotchner", lastName: "Maria" },
            { firstName: "Phantom", lastName: "Lancer" },
        ])
        .execute()
        console.log(`Nombre de tuples insérés : ${count.raw.affectedRows}`);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
INSERT INTO USER (firstName, lastName) VALUES ("Hotchner", "Maria"), ("Phantom", "Lancer")
```

TypeORM

Pour modifier

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .update(User)
        .set({ firstName: "Timber", lastName: "Saw" })
        .where("id = :id", { id: 1 })
        .execute()
    console.log(`Nombre de tuples modifiés : ${count.affected}`);

}).catch(error => console.log(error))
```

TypeORM

Pour modifier

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .update(User)
        .set({ firstName: "Timber", lastName: "Saw" })
        .where("id = :id", { id: 1 })
        .execute()
    console.log(`Nombre de tuples modifiés : ${count.affected}`);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
UPDATE USER SET firstName = "Timber", lastName = "Saw" WHERE id = 1
```

TypeORM

Pour supprimer

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .delete()
        .from(User)
        .where("id = :id", { id: 1 })
        .execute()
        console.log(`Nombre de tuples supprimés : ${count.affected}`);

}).catch(error => console.log(error))
```

TypeORM

Pour supprimer

```
import { AppDataSource } from "./data-source"
import { User } from "./entity/User"

AppDataSource.initialize().then(async () => {

    const userRepository = AppDataSource.getRepository(User);
    const count = await userRepository
        .createQueryBuilder("user")
        .delete()
        .from(User)
        .where("id = :id", { id: 1 })
        .execute()
        console.log(`Nombre de tuples supprimés : ${count.affected}`);

}).catch(error => console.log(error))
```

Requête SQL exécutée

```
DELETE FROM user WHERE id = 1
```