WILD CODE SCHOOL

TypeORM

Les entités

Qu’est-ce qu’une ORM ?

Comment l’utiliser ?

ORM

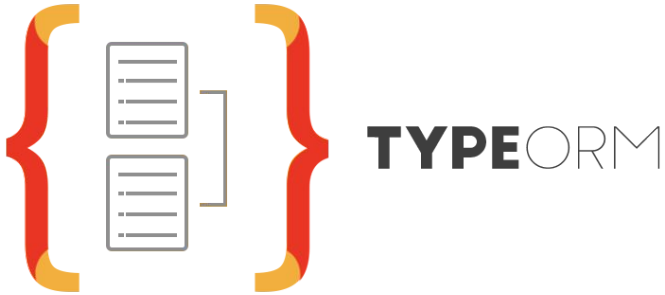
- **ORM** : Object-relational mapping
- Lié au paradigme de la POO
- Manipulation des données de la BDD à l'aide d'objets
- Pas besoin d'écrire de requêtes SQL
- Une table est représentée par une **classe d'entité**
- Plus compréhensible, plus facile à maintenir
- Requêtes possiblement moins optimisées si pas utilisées correctement

ORM

- Exemples d'ORM :
  - Hibernate en Java
  - Doctrine en PHP
  - Eloquent en PHP
  - SQLAchemy en Python
  - Sequelize en Javascript
  - ...

TypeORM

- NodeJS
- Support de Typescript
- Fonctionne avec des décorateurs @
- Active Record ou Data Mapper
- Basé sur Hibernate (Java) et Doctrine (PHP)



TypeORM : installation

Installer le package NPM

Modifier le fichier de configuration de typescript pour supporter les décorateurs

```
$ npm install typeorm
$ npm install reflect-metadata
```

tsconfig.json

```
{
  …,
  "emitDecoratorMetadata": true,
  "experimentalDecorators": true,
  …
}
```

Datasource : SQLite

Définir les **informations de connexion** à la base.

Définition du type de la base.

Définition du chemin vers la base.

src/config/db.ts

```
import { DataSource } from "typeorm";
import { Student } from "./entities/student;
import { School } from "./entities/school;
import { Language } from "./entities/language;

export const dataSource = new DataSource({
  type: "sqlite",
  database: "../../wild.sqlite",
  entities: [Student, School, Language],
  synchronize: true,
});
```

Datasource

Renseigner les entités une à une.

src/config/db.ts

```
import { DataSource } from "typeorm";
import { Student } from "./entities/student;
import { School } from "./entities/school;
import { Language } from "./entities/language;

export const dataSource = new DataSource({
  type: "sqlite",
  database: "../../wild.sqlite",
  entities: [Student, School, Language],
  synchronize: true,
});
```

Datasource

Renseigner toutes les entités en même temps.

src/config/db.ts

```
import { DataSource } from "typeorm";
import { Student } from "./entities/student;
import { School } from "./entities/school;
import { Language } from "./entities/language;

export const dataSource = new DataSource({
  type: "sqlite",
  database: "../../wild.sqlite",
  entities: ["src/entities/*.ts"],
  synchronize: true,
});
```

Datasource

Les modifications apportées aux entités sont appliqués sur la structure de la base automatiquement.

Ne jamais utiliser en production.

src/config/db.ts

```
import { DataSource } from "typeorm";
import { Student } from "./entities/student;
import { School } from "./entities/school;
import { Language } from "./entities/language;

export const dataSource = new DataSource({
  type: "sqlite",
  database: "../../wild.sqlite",
  entities: ["src/entities/*.ts"],
  synchronize: true,
});
```

Datasource : Postres

Une base de données SQLite a seulement besoin du chemin du fichier contenant la base.

MySQL ou encore Postgres ont **besoin d'informations supplémentaires.**

Exemple de connexion à Postgres.

src/config/db.ts

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

export const dataSource = new DataSource({
    type: "postgres",
    host: "localhost",
    port: 5432,
    username: "test",
    password: "test",
    database: "test",
    synchronize: true,
    logging: true,
    entities: [Post, Category],
});
```

Initialisation

Initialisation de la datasource en même temps que le serveur.

TypeORM a besoin de la librairie reflect-metadata avec ses décorateurs afin de construire les requêtes SQL.

src/index.ts

```
import "reflect-metadata";
import express from 'express';

…

app.listen(5001, async () => {
  await dataSource.initialize();
  console.log('Server launch on
http://localhost:5001');
});
```

Entités

Une entité TypeORM est une classe qui représente une table dans la base de données

Définition d'une entité avec le décorateur ***@Entity***

Définition d'une colonne avec le décorateur ***@Column({options})***

TypeORM détermine le type de données automatiquement en fonction du type typescript.

Possibilité de définir des contraintes dans les options d'une colonne.

https://typeorm.io/entities#column-options

src/entities/student.ts

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

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

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

  @Column()
  name: string;

  @Column()
  birthday: Date;

  @Column()
  address: string;
}
```

Active Record

Un active record est un **pattern de programmation** qui défini que :

*Chaque enregistrement de la base est représenté par un objet qui **contient les données** elles-mêmes mais aussi **tous les comportements** (méthodes) lié à ces données (select, insert, update, remove).*

Ceci est possible grâce à l'**héritage de la classe BaseEntity**.

src/entities/student.ts

```ts
import {
  BaseEntity,
  Column, Entity,
  PrimaryGeneratedColumn
} from "typeorm";

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

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

  @Column()
  name: string;

  @Column()
  birthday: Date;

  @Column()
  address: string;
}
```

Création

On instancie un **nouvel objet.**

On renseigne ses attributs.

On appelle la **méthode save() héritée** de la classe **BaseEntity**

Le comportement de sauvegarde de l'entité est appelé directement sur l'entité car c'est un Active Record.

src/index.ts

```
…

app.post("/student", (req, res) => {
  const ad = new Student();
  ad.firstname = req.body.firstname;
  ad.name = req.body.name;
  ad.birthday = req.body.birthday;
  ad.address = req.body.address;

  ad.save();

  res.send(ad);
});

…
```

Modification

On récupère d'abord l'objet existant. Cela permet de gérer une erreur dans le cas où l'enregistrement n'existe pas.

On assigne les nouvelles valeurs aux attributs de l'objet à modifier.

On appelle la **méthode save() héritée** de la classe **BaseEntity**

src/index.ts

```
…

app.put("/ad/:id", async (req, res) => {
  const id = parseInt(req.params.id);
  const st = await Student.findOneBy({ id })
  if (st !== null) {
    st.firstname = req.body.firstname;
    st.name = req.body.name;
    st.birthday = req.body.birthday;
    st.address = req.body.address;
    st.save();
  }
  res.send(st);
});

…
```

Suppression

Suppression possible directement en appelant la **méthode de classe delete()** directement sur la classe d'entité.

Suppression possible également via la **méthode remove() héritée de BaseEntity**.

src/index.ts

```
…

app.delete("/ad/:id", async (req, res) => {
  const id = parseInt(req.params.id);
  await Student.delete({ id });
  res.send('OK');
});

// ou
app.delete("/ad/:id", async (req, res) => {
  const id = parseInt(req.params.id);
  const st = Student.findOneBy({ id });
  if (st !== null) {
    st.remove();
  }
  res.send('OK');
});

…
```

Validation

Possible d'utiliser **class-validator** pour valider les données d'une entité avant insertion en base.

```
$ npm install class-validator
```

https://github.com/typestack/class-validator

src/entities/city.ts

```
import {
  BaseEntity,
  Column, Entity,
  PrimaryGeneratedColumn } from "typeorm";
import { Length, Min, Max } from
"class-validator";

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

  @Column({ length: 100 })
  @Length(10, 100, {
    message: "Entre 10 et 100 caractères"
  })
  city: string;

  @Column()
  @Min(10)
  @Max(80)
  capacity: number;
}
```

Validation

La validation se fait manuellement en appelant la **fonction validate** de class-validator.

Exemple

```
import { validate } from "class-validator";

const student = new Student();
student.firstname = "Marc";
student.name = "Dupont";
student.save();

const errors = await validate(post)
if (errors.length > 0) {
    throw new Error(`Validation failed!`)
} else {
    await dataSource.manager.save(post)
}
```

Récupération

Récupérer plusieurs enregistrements.

La méthode **find(options)** renvoie tous les enregistrements de la table.

Pour effectuer un filtre, privilégier la méthode **findBy(filters, options)**

La structure filters peut contenir plusieurs méthodes TypeORM :

- Not
- LessThan
- LessThanOrEqual
- Like
- ...

https://typeorm.io/find-options

Exemples

```
import { LessThan, Like } from "typeorm"

const students = await Student.find();

const students = await Student.findBy({
      name: "Rob"
});

const schools = await School.findBy({
      capacity: LessThan(40)
});

const schools = await School.findBy({
      city: Like("%ar%")
});
```

Récupération

Récupérer un seul enregistrement.

La méthode **findOneBy(filters)** renvoie le 1er enregistrement.

https://typeorm.io/find-options

src/index.ts

```
…

app.get("/ad", async (req, res) => {
  const ads = await Student.find();
  res.send(ads);
});

app.get("/ad/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const ad = Ad.findOneBy({ id });
  res.send(ad);
});

…
```

Transactions

TypeORM permet de gérer les transactions.

Une transaction est une fonction avec une callback.

```
await dataSource.transaction(async (transactionalEntityManager) => {
  // Vos requêtes
});


// OU

const queryRunner = dataSource.createQueryRunner();
await queryRunner.connect();
await queryRunner.startTransaction();
// Vos requêtes
await queryRunner.commitTransaction();
await queryRunner.rollbackTransaction();
await queryRunner.release();
```

Index

Décorateur **@Index()**

src/entities/student.ts

```
import {
  BaseEntity,
  Column, Entity,
  PrimaryGeneratedColumn,
  Index
} from "typeorm";

@Entity()
@Index(["firstname", "birthday"])
export class Student extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number;

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

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

  @Column()
  birthday: Date;

  @Column()
  address: string;
}
```

Entity Listener

Possibilité d'exécuter des méthodes de la classe d'entité suivant certains **évènements**.

Assignation de la date de création juste avant l'insertion.

https://typeorm.io/listeners-and-subscribers

src/entities/ad.ts

```
import { …, BeforeInsert } from "typeorm";
import { Category } from "./category";

@Entity()
export class Ad extends BaseEntity {
  …
  @Column()
  location: string;

  @ManyToOne(() => Category)
  category: number;

  @Column()
  createdAt: Date;

  @BeforeInsert()
    updateDates() {
        this.createdAt = new Date();
    }
}
```

Atelier

The good corner

The Good Corner

Nous allons remplacer l'utilisation du driver SQLite par TypeORM.

Nous allons nous concentrer sur la table Annonce (Ad) sans gérer la liaison catégorie.

- Créer la datasource
- Initialiser la datasource dans index.ts
- Créer l'entité Ad
- Lier l'entité à la datasource si ce n'est pas fait automatiquement
- Modifier le endpoint GET pour utiliser TypeORM
- Modifier le endpoint POST pour utiliser TypeORM
- Modifier le endpoint PUT pour utiliser TypeORM
- Modifier le endpoint DELETE pour utiliser TypeORM