Méthode de connexion alternative à la base de données

Utilisation des lignes de commandes

Création de la base de données

Pour faire simple et éviter les messages d'erreur, nous allons initialiser un nouveau projet symfony dans notre dossier `Symfony` (méthode vu précédemment). Appelons ce projet `exempleDoctrine`.

Une fois le projet crée, allons de suite dans le fichier `.env` pour paramétrer la connexion à notre base de données. Nous resterons sur la base northwind, pensez donc à la supprimer de votre SGBD.

Le fichier `.env` paramétré correctement, nous pouvons créer la base de données.

Ouvrez un terminal de commande dans le dossier racine de votre projet, ou utilisez celui de votre IDE et taper la commande suivante :

```
php bin/console doctrine:database:create
```

La console nous dit que la basse de données `northwind` a bien été créée :

En vérifiant dans phpmyadmin (ou autre SGBD), on remarque bien qu'une base vide a été créée.

Création de la table `products`

Toujours dans l'invite de commande, saisissez la commande suivante :

```
php bin/console make:entity
```

La console nous demande le nom de l'entité (le nom de la table que nous voulons créer) :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:entity

 Class name of the entity to create or update (e.g. OrangeChef):
 > 
```

Créons la table `products` :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:entity

 Class name of the entity to create or update (e.g. OrangeChef):
 > products
products

  created: src/Entity/Products.php
  created: src/Repository/ProductsRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this command.

 New property name (press <return> to stop adding fields):
 > 
```

On voit qu'un fichier a été créé dans le dossier `Entity`, et un autre dans le dossier `Repository`.

La console nous demande de renseigner les différentes propriétés de notre table :

- `ProductName`
- `CategoryID`
- `QuantityPerUnit`
- `UnitPrice`
- `UnitsInStock`
- `UnitsOnOrder`
- `ReorderLevel`
- `Discontinued`

Dans un premier temps, nous ne renseignons pas les clés étrangères (ici `supplierID`), nous verrons ça un peu plus tard lors de l'établissement des relations entre les tables.

Pour chaque propriété entrée, la console nous demande certaines informations (si vous avez un doute, tapez `?` pour plus d'infos) :

- le type de champs (int, string, text, ...)
- la taille du champs
- si le champs est nullable en base de données ou non

On a ensuite le choix d'ajouter une autre propriété ou non, dans ce cas, appuyez sur `entrée` pour finaliser l'action.

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:entity

 Class name of the entity to create or update (e.g. Oran
geChef):
 > products
 created: src/Entity/Products.php
 created: src/Repository/ProductsRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this command.

 New property name (press <return> to stop adding fields):
 > ProductName

 Field type (enter ? to see all types) [string]:
 > string
strin
g

 Field length [255]:
 > 40

 Can this field be null in the database (nullable) (yes/no) [no]:
 > no

 updated: src/Entity/Products.php

 Add another property? Enter the property name (or press <return> to
stop adding fields):
 > |
```

L'`id` de la table (ici `ProductId`) est automatiquement créé lors de la création de la table.

Finalisez la création de la table `products`.

Toujours dans la console, exécuter la commande suivante :

```
php bin/console make:migration
```

Vous observerez qu'un fichier de migration a été créé,

Ouvrez ce fichier, et observez ce qu'il s'y trouve.

Les fichiers de migrations recoupent toutes les informations concernant les modifications apportées à la base de données. Vous pouvez vérifier ces informations, les modifier si nécessaire.

Une fois que vous avez terminé votre vérification, vous pouvez taper dans la console la commande suivante :

```
php bin/console doctrine:migrations:migrate
```

La console nous avertit que nous allons effectuer des modifications dans base de données, et nous demande une confirmation.

Confirmez, et observez les modifications dans la base de données.

Pour créer une entité, vous pouvez aussi spécifier directement le nom de l'entité lorsque vous taper la ligne de commande :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:entity customers
```

Reproduisez cette opération pour les autres tables de la base Northwind pour finir la création de la base. Ne mettez pas les clés étrangères lors de la création de vos tables, sous peine d'avoir des soucis lors de la mise en place de vos relations avec Doctrine.

Modification d'une entité

Pour modifier une entité, il suffit de refaire `php bin/console make:entity` avec le nom de l'entité à changer :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:entity products

 Your entity already exists! So let's add some new fields!

 New property name (press <return> to stop adding fields):
 > 
```

On peut ensuite ajouter les champs dont on a besoin.

Les relations entre les tables

Doctrine permet la mise en place des relations entre les tables de manière assez simple. Pour cela il suffit de créer dans la table correspondante le champ qui correspondra à la clé étrangère.

Prenons ici pour exemple la relation entre `products` et `suppliers`.

Nous devons ajouter une clé étrangère dans la table `products` pour la relier à la table `suppliers`.

On fait une modification d'entité (cf ci-dessus), et on ajoute le champ `SupplierID`.

Lorsque la console nous demande de renseigner le type de champ, y inscrire `relation`. Il faut ensuite renseigner avec quelle **classe** nous faisons cette relation, ici `Suppliers`.

Ensuite renseignons quel type de relation se fait entre nos tables. la console nous propose 4 types de relations, avec en prime une petite explication pour chacune d'elles :

```
 New property name (press <return> to stop adding fields):
 > SupplierID

 Field type (enter ? to see all types) [string]:
 > relation
relatio
n

 What class should this entity be related to?:
 > Suppliers
Suppliers

What type of relationship is this?
------------ --------------------------------------------------------------------
 Type         Description
------------ --------------------------------------------------------------------
 ManyToOne    Each Products relates to (has) one Suppliers.
              Each Suppliers can relate to (can have) many Products objects

 OneToMany    Each Products can relate to (can have) many Suppliers objects.
              Each Suppliers relates to (has) one Products

 ManyToMany   Each Products can relate to (can have) many Suppliers objects.
              Each Suppliers can also relate to (can also have) many Products objects

 OneToOne     Each Products relates to (has) exactly one Suppliers.
              Each Suppliers also relates to (has) exactly one Products.
------------ --------------------------------------------------------------------

 Relation type? [ManyToOne, OneToOne, ManyToMany, OneToOne]:
 > |
```

Il ne reste plus qu'à choisir celle qui nous convient (ici `ManyToOne`).

Nous devons ensuite renseigner si la propriété est nullable ou non (ici oui).

Doctrine propose ensuite d'ajouter une nouvelle propriété dans la **classe** `Suppliers` qui nous permettrais d'avois accès directement aux produits. C'est-à-dire que pour un fournisseur par exemple, on pourrait récupérer tous ses produits à l'aide de `$suppliers->getProducts`. Cela peut être assez intéressant et surtout faire gagner du temps selon les fonctionnalités à développer sur le site. (mettons 'out' ici, la console nous demande le nom pour cette nouvelle propriété).

Une fois la relation faite, il reste à faire la migration comme vu précédemment.

Création d'un contrôleur

Plutôt de créer un contrôleur manuellement, nous pouvons utiliser la console pour générer un contrôleur sur une table, en utilisant la commande suivante :

```
php bin/console make:controller
```

Lors du nommage du contrôleur, nous respecterons la convention en mettant une majuscule au début, suivis de 'Controller'. Exemple pour la création d'un contrôleur pour `Suppliers` :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:controller

 Choose a name for your controller class (e.g. VictoriousElephantController):
 > SuppliersController

  created: src/Controller/SuppliersController.php
  created: templates/suppliers/index.html.twig


  Success!


 Next: Open your new controller class and add some pages!

Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ |
```

On observe donc qu'un contrôleur est créé, ainsi qu'une vue.

Gestion du Crud

De la même façon qu'un contrôleur, nous pouvons créer facilement et rapidement un CRUD complet (mais basique) sur une entité (table).

Pour cela utilisons la commande `php bin/console make:crud` sur l'entité `Products` :

```
Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ php bin/console make:crud

 The class name of the entity to create CRUD (e.g. OrangeJellybean):
 > Products
Product
s

  created: src/Controller/ProductsController.php
  created: src/Form/ProductsType.php
  created: templates/products/_delete_form.html.twig
  created: templates/products/_form.html.twig
  created: templates/products/edit.html.twig
  created: templates/products/index.html.twig
  created: templates/products/new.html.twig
  created: templates/products/show.html.twig


  Success!


 Next: Check your new CRUD by going to /products/

Cédric Cousin-RubyRDESKTOP-AVTSD9C MINGW64 /c/wamp64/www/Symfony/exempleDoctrine
$ |
```

Nous observons la création de plusieurs fichier :

- un contrôleur
- un formType (ProductsType) qui sert à la génération des formulaires par Symfony
- et un ensemble de templates qui correspondent aux différentes vues nécessaires à notre CRUD.

Rendez-vous sur la route du `ProductsController` afin de profiter du résultat, ajoutez auparavant Bootstrap dans `base.html.twig` afin d'avoir un affichage un peu plus propre (pour rappel, Symfony utilise par défaut des classes de Bootstrap).