Gestion simplifiée de recettes

L’objectif de ce TP est de créer une version simplifiée d’une application pour des recettes de cuisine.

Avant de commencer, voici une petite liste de quelques commandes disponibles quand le dossier bin > console qui seront utiles pour ce TP.

```
  doctrine:database:create               Creates the configured database
  doctrine:database:drop                 Drops the configured database
  doctrine:fixtures:load                 Load data fixtures to your database
  doctrine:migrations:current            Outputs the current version
  doctrine:migrations:execute            Execute one or more migration versions up or
down manually.
  doctrine:migrations:list               Display a list of all available migrations and
their status.
  doctrine:migrations:migrate            Execute a migration to a specified version or
the latest available version.
  doctrine:migrations:rollup             Rollup migrations by deleting all tracked
versions and insert the one version that exists.
  doctrine:schema:create                 Processes the schema and either create it
directly on EntityManager Storage Connection or generate the SQL output
  doctrine:schema:drop                   Drop the complete database schema of
EntityManager Storage Connection or generate the corresponding SQL output
  doctrine:schema:update                 Executes (or dumps) the SQL needed to update
the database schema to match the current mapping metadata
  doctrine:schema:validate               Validate the mapping files

  make:auth                              Creates a Guard authenticator of different
flavors
  make:controller                        Creates a new controller class
  make:docker:database                   Adds a database container to your
docker-compose.yaml file
  make:entity                            Creates or updates a Doctrine entity class,
and optionally an API Platform resource
  make:fixtures                          Creates a new class to load Doctrine fixtures
  make:migration                         Creates a new migration based on database
changes
  make:registration-form                 Creates a new registration form system
  make:security:form-login               Generate the code needed for the form_login
authenticator
```

Partie 0 : Configurations

Cloner le répertoire suivant :

```
git clone https://github.com/biveguengoa/recipes.git
```

Dans le répertoire recipes, installez les dépendances :

```
composer install
```

Configuration de la base de données

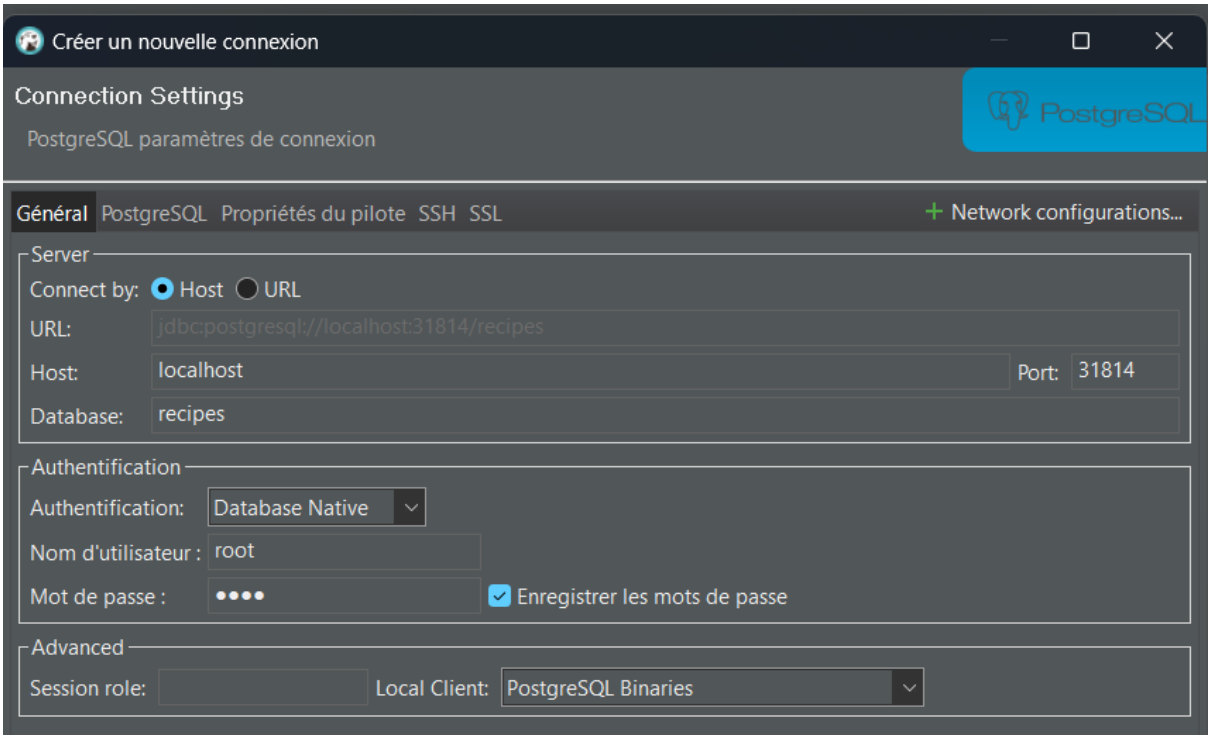
Pour simplifier le travail, une bonne partie des configurations a déjà été faite dans le répertoire que vous avez cloné. Nous allons utiliser Docker, PostgreSQL ainsi que DBeaver pour pouvoir manipuler la base de données du projet.

Tout d'abord, lancez Docker en arrière plan :

```
docker-compose up -d
```

Vérifiez que tout fonctionne bien avant de continuer : `docker-compose ps`
Dans le fichier `.env`, modifiez le port de la variable DATABASE_URL par le numéro qui a été renvoyé par la commande `docker-compose ps`.

Accéder à la base de données dans DBeaver en ajoutant une nouvelle connexion comme suit : Base de données > Nouvelle connexion > PostgreSQL



⚠️Pensez à bien modifier le numéro de port

Une fois toutes les configurations faites, vous pouvez lancer le serveur : `symfony server:start` et y accéder à l'adresse suivante : http://localhost:8000/.

Partie 1 : Hello world

En accédant à l'application, on constate qu'il y a une erreur de type **404 Not Found**. C'est normal car il n'y a encore aucune page associée.

L'objectif de cette partie est donc de créer une première page qui affiche `Hello world`. Pour cela, nous allons suivre 3 étapes :

1. Créer un controller `HomeController` pour gérer les requêtes
2. Associer une route à la méthode `index` dans le controller
3. Créer une page HTML `home.html.twig` dans le dossier template. Ce template, comme tous les templates que nous créerons durant le tp, devra étendre `base.html.twig` comme suit : `{% extends "base.html.twig" %}`

Pour le design, nous utiliserons Bootswatch united mais dans cette partie il n'y a rien à gérer.

Partie 2 : Première entité et CRUD

Nous allons créer la première entité de notre application et réaliser les opérations CRUD sur cette entité.

1. Création de l'entité Ingredient

À l'aide de la commande `php bin/console make:entity`, créez une entité Ingredient qui a pour propriétés : un nom et une date de création (aucune de ces propriétés ne doit être nulle). Initialisez la date de création dans le constructeur.

La création de l'entité a généré un répertoire `repository` composé d'une classe IngredientRepository. En survolant rapidement la classe, on comprend rapidement que le principe est le même que celui d'un repository en Spring Boot.

1.1. Migration

Ajoutez la table correspondant à l'entité Ingredient en utilisant les bonnes commandes.

1.2. Validation

La validation est une tâche très courante dans les applications Web. Pour éviter d'avoir des chaînes de caractères de longueur inférieures à 2 ou encore des entiers nuls (=0), nous allons rajouter des contraintes sur les données avant qu'elles soient chargées dans la base de données.

Dans la classe `Ingredient`, rajoutez la contrainte suivante : le nom doit avoir une longueur comprise entre 2 et 50.

⚠️Pensez toujours à faire une migration à chaque fois que vous apportez des modifications sur une entité.

1.3. Premiers ingrédients

Nous remplirons plus tard la base de données avec de vraies données. Pour l'instant, nous nous contenterons d'alimenter la bdd avec de "fausses" données. Nous utiliserons les Fixtures prévus à cet effet par Symfony.

Commencez par installer les dépendances nécessaires :

```
composer require --dev orm-fixtures
```

La commande a généré une classe AppFixtures dans le dossier `src>DataFixtures` composée d'une méthode load pour charger les données dans la base de données.

Modifiez le corps de cette méthode pour créer 10 ingrédients puis chargez-les dans la base de données. Une fois le chargement effectué, vérifiez le contenu de la table dans dbeaver.

❗ **Pensez à faire persister l'objet à l'aide de la méthode persist de ObjectManager.**

2. CRUD

2.1. Lecture

Nous allons commencer par créer un controller IngredientController associé à l'entité Ingredient :

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

En plus du controller, la commande a également généré un template `template > ingredient > index.twig.html` pour l'affichage des informations concernant les ingrédients .

Faites les modifications nécessaires pour afficher l'ensemble des ingrédients.

Ingredients

Numéro	Nom
1	Ingredient 1
2	Ingredient 2
3	Ingredient 3
4	Ingredient 4
5	Ingredient 5
6	Ingredient 6
7	Ingredient 7
8	Ingredient 8
9	Ingredient 9
10	Ingredient 10

💡 Vous pouvez passer par un EntityManager ou simplement utiliser l'injection de dépendance comme ici : DB & Doctrine ORM pour récupérer les données dans la base de données.

2.2. Création

À présent, nous voulons ajouter des ingrédients à partir d'un formulaire. Comme vu dans le cours, Symfony inclut une fonctionnalité de formulaire assez riche facilitant la création et le traitement des formulaires.

Implémentez un formulaire IngredientType pour gérer la création d'un ingrédient.

❗ Workflow pour les formulaires :

1. Construire le formulaire dans une classe dédiée ;
2. Afficher le formulaire dans un template ;
3. Traiter le formulaire pour la validation des données soumises.

Pour les différents champs du formulaire, ajoutez les options suivantes pour vérifier les données saisies :

- Nom :

```
[
                'attr' => ['class' => 'form-control'],
                'label' => 'Name',
                'label_attr' => [
                    'class'=> 'form-label mt-4',
                    'minlength' => '2',
                    'maxlength' => '50' ],
                'constraints' => [
                    new Assert\NotBlank(),
                    new Assert\Length(['min' => 2, 'max' => 50])
]
```

- Submit

```
[
'attr' => ['class' => "btn btn-primary mt-4"],
                'label' => 'Create ingredient'
]
```

Pour un meilleur visuel, ajoutez dans le template de création les lignes de codes suivantes :

```
{{ form_start(form) }}
            <div class="form-group">
```

```
                {{ form_label(form.name) }}
                {{ form_widget(form.name) }}
                <div class="form-error">
                    {{ form_errors(form.name) }}
                </div>
            </div>
            <div class="form-group">
                {{ form_row(form.save) }}
            </div>
{{ form_end(form) }}
```

2.3. Modification

Le principe ici est le même que pour la création.

Etant donné qu'on a déjà fait une partie du travail dans l'étape précédente, il faudra simplement créer un template pour le formulaire de modification (pour aller plus vite, reprendre simplement celui du Create et le réadapter) et ajouter une méthode dans le controller.
La méthode edit prendra en plus en paramètre un Ingredient qui sera passé directement à la fonction `createForm`.

2.4. Suppression

Définissez dans le controller une méthode delete qui prend en paramètre un `EntityManagerInterface` et un `Ingredient` et renvoie une objet de type response.

2.5. Flash message

Pour le moment, lorsqu'on soumet un formulaire ou supprime des données on n'a aucune indication quant au déroulement de l'action. Pour changer cela, nous allons utiliser les `flash message` de Symfony pour afficher des messages de succès/erreur.

Ajoutez dans `IngredientController` l'appel de méthode suivant quand c'est nécessaire :

```
$this->addFlash(type, message); //type = success ou error
```

Dans le fichier index.html.twig, nous allons utiliser une boucle for pour afficher le contenu du flash message : `{% for message in app.flashes(type) %}`

Partie 3 : Les recettes et les relations entre entités

Nous allons reprendre les mêmes étapes que précédemment pour gérer les recettes.

Une recette est définie par un nom (ou titre), une durée (en minute), un nombre de personnes, un niveau de difficulté (de 1 à 5), une description (ou étapes), une date de création ainsi qu'une liste d'ingrédients (également initialisée dans le constructeur). Seules les propriétés nom, description et ingrédients sont obligatoires, les autres quant à elles, peuvent être nulles.

Pour gagner du temps, nous vous fournissons le code du template.

Il n'est pas nécessaire d'utiliser les Fixtures dans cette étape mais si vous souhaitez avoir un premier visuel, vous pouvez utiliser la même logique que pour les ingrédients.

1. Relations entre entités

La propriété $ingredients est de type Collection. Cette propriété porte l'attribut `#[ORM\ManyToMany(targetEntity: Ingredient::class)]` indiquant que les entités Recipe et Ingredient ont une relation de type **Many-To-Many unidirectionnelle** i.e. que l'on s'intéressera uniquement au fait qu'une recette ait plusieurs ingrédients.

2. Formulaire de création

L'implémentation du formulaire pour la création d'une recette est similaire à celle des ingrédients.

Pour l'instant, si on essaie d'accéder au formulaire de création, il y aura probablement une erreur car le navigateur ne saura pas comment afficher des objets. Nous allons corriger cette erreur en ajoutant d'abord dans la classe `Ingredient` une méthode `__toString()` qui retourne le nom de l'ingrédient. Ensuite, dans la classe `IngredientType`, définissez le type du champ ingrédient en `EntityType::class`.

2.1. Query builder

Les modifications faites précédemment ne suffisent pas. Nous allons rajouter quelques options au champ.

```
[
 'class' => Ingredient::class,
 'query_builder' => function (IngredientRepository $r) {
                        return $r->createQueryBuilder('i')
                            ->orderBy('i.name', 'ASC');
                    },
 'label' => 'Ingredients',
 'label_attr' => ['class'=> 'form-label mt-4'],
 'choice_label' => 'name',
 'multiple' => 'true',
 'expanded' => 'true'
]
```

L'option 'query_builder' permet définir une requête personnalisée pour récupérer les données d'une entité dans un formulaire ou ailleurs. Sa valeur est une fonction anonyme qui prend un en paramètre un objet `IngredientRepository` et utilise le Query Builder [1]de Doctrine pour construire une requête :

- La requête construite sélectionne tous les enregistrements de la table d'entité associée à `IngredientRepository` (ingrédient), puis les trie par nom dans l'ordre alphabétique.

[1] En Symfony, un "query builder" est une fonctionnalité qui vous permet de construire des requêtes SQL de manière programmatique à l'aide d'une API orientée objet, plutôt que d'écrire directement des requêtes SQL en chaînes de caractères.