Bibliothèque en ligne (version simplifiée)

L'objectif de ce TP est de créer une version simplifiée d'une application pour emprunter des livres. Dans cette application, les utilisateurs auront la possibilité d'emprunter des livres et de proposer leurs propres livres à d'autres utilisateurs.

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:ensure-production-settings    Verify that Doctrine is properly configured for a production environment
  doctrine:fixtures:load                 Load data fixtures to your database
  doctrine:migrations:current            Outputs the current version
  doctrine:migrations:diff               Generate a migration by comparing your current database to your mapping information.
  doctrine:migrations:dump-schema        Dump the schema for your database to a migration.
  doctrine:migrations:execute            Execute one or more migration versions up or down manually.
  doctrine:migrations:generate           Generate a blank migration class.
  doctrine:migrations:latest             Outputs the latest version
  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:migrations:status             View the status of a set of migrations.
  doctrine:migrations:sync-metadata-storage  Ensures that the metadata storage is at the latest version.
  doctrine:migrations:up-to-date         Tells you if your schema is up-to-date.
  doctrine:migrations:version            Manually add and delete migration versions from the version table.
  doctrine:schema:create                 Processes the schema and either create it directly on EntityManager Storage Connection or ger
  doctrine:schema:drop                   Drop the complete database schema of EntityManager Storage Connection or generate the corresp
  doctrine:schema:update                 Executes (or dumps) the SQL needed to update the database schema to match the current mapping
  doctrine:schema:validate               Validate the mapping files

 make
  make:auth                              Creates a Guard authenticator of different flavors
  make:command                           Creates a new console command class
  make:controller                        Creates a new controller class
  make:crud                              Creates CRUD for Doctrine entity 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:form                              Creates a new form class
  make:message                           Creates a new message and handler
  make:messenger-middleware              Creates a new messenger middleware
  make:migration                         Creates a new migration based on database changes
  make:registration-form                 Creates a new registration form system
  make:reset-password                    Create controller, entity, and repositories for use with symfonycasts/reset-password-bundle
  make:security:form-login               Generate the code needed for the form_login authenticator
  make:test                              [make:unit-test|make:functional-test] Creates a new test class
  make:twig-component                    Creates a twig (or live) component
  make:twig-extension                    Creates a new Twig extension with its runtime class
```

Partie 0 : Configurations

Cloner le répertoire suivant :

```
git clone https://github.com/biveguengoa/library.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 utiliserons Docker, PostgreSQL ainsi que DBeaver pour pouvoir manipuler la base de données du projet.

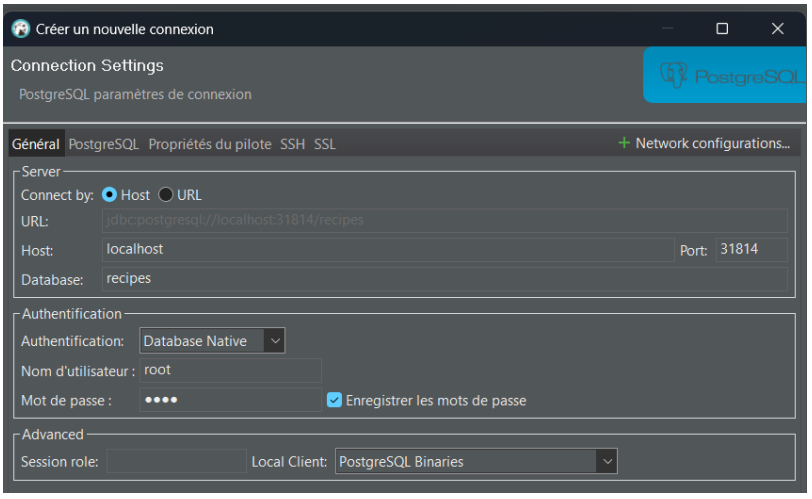
Tout d'abord, lancez Docker en arrière plan :

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 Book

À l'aide de la commande php bin/console make:entity, créez une entité Ingredient ayant pour propriétés :

- un titre ;
- une catégorie (éventuellement plusieurs) ;
- une disponibilité (pour savoir si le livre est déjà emprunté ou non) ;

- une description ;
- une date d’emprunt.

Seules la description et la date peuvent avoir des valeurs nulles.

La création de l’entité a généré un répertoire repository composé d’une classe BookRepository. 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

Nous allons utiliser la migration pour charger nos différentes entités dans la base de données. En symfony, les migrations servent à gérer les modifications de la structure de la base de données en fonction des changements dans le code de l'application.

Ajoutez puis chargez la table correspondant à l’entité Book en utilisant les bonnes commandes.

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 Book, ajoutez les contraintes suivantes :

- Le titre ne doit pas être vide et sa longueur est comprise entre 3 et 50.
- Au moins une catégorie doit être sélectionnée et être au moins d’un des types suivants : Science-Fiction, Mystère/Thriller, Romance, Fantasy, Poésie, Non-Fiction, Aventure, Manga, BD, Autres. Vous pouvez vous servir de la documentation pour trouver la bonne contrainte : Validation.
- La disponibilité ne doit pas être nulle.

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

Premiers livres

Nous remplirons plus tard la base de données avec de vraies données. Pour l’instant, nous nous contenterons de l’alimenter avec de “fausses” données. Nous utiliserons le composant prévu à cet effet par Symfony : les Fixtures.

Commencez par installer les dépendances nécessaires :

```
composer require --dev orm-fixtures
composer require fakerphp/faker --dev
```

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 20 livres 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.**

▼ Solution

```
protected Generator $faker;

public function load(ObjectManager $manager): void {
        $categories = ["Science-Fiction", "Mystère/Thriller", "Romance", "Fantasy", "Poésie",
        "Non-Fiction", "Aventure", "Manga", "BD", "Autres"];

        for ($i = 0; $i < 20; $i++) {
            $book = new Book();
            $book->setTitle($this->faker->words((mt_rand(3, 4)), true));
```

```
            $book->setAvailable(true);
            $book->setDescription($this->faker->realTextBetween(100, 300));
            $book_cat = [];
            for($j = 0; $j < (mt_rand(1, 3)); $j++) {
                $book_cat[] = $categories[mt_rand(0, count($categories) -1)] ;
            }
            $book->setCategories($book_cat);

            $manager->persist($book);
        }

        $manager->flush();
    }
```

CRUD

Lecture

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

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

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

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

Livres

Titre	Categorie	Disponibilité
Livre 0	Science-Fiction, Poésie	Oui
Livre 1	Fantasy	Oui
Livre 2	Fantasy, Poésie	Oui
Livre 3	Autres, Romance	Oui
Livre 4	Mystère/Thriller, Romance	Oui
Livre 5	Romance, Manga	Oui
Livre 6	Fantasy, Non-Fiction, BD	Oui
Livre 7	Fantasy, Science-Fiction	Oui
Livre 8	BD, Aventure, Mystère/Thriller	Oui
Livre 9	Romance, Science-Fiction	Oui
Livre 10	Mystère/Thriller, Autres	Oui

> 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.

Création

À présent, nous voulons ajouter des livres 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 BookType pour gérer la création d'un livre.

▼ **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 :

- Titre :

```
[
    'attr' => [
        'class' => 'form-control',
        'minlength' => '3',
        'maxlength' => '50'
    ],
    'label' => 'Title',
    'label_attr' => ['class'=> 'form-label mt-4'],
    'constraints' => [
        new Assert\NotBlank(),
        new Assert\Length(['min' => 3, 'max' => 50])
    ]
]
```

- Catégorie :

```
[
    'choices'  => Book::CATEGORIES,
    'choice_attr' => function() {
        return ['class' => 'form-check-input me-1 ms-3'];
    },
    'label' => 'Catégorie(s)',
    'label_attr' => ['class'=> 'form-label mt-4'],
    'multiple' => 'true',
    'expanded' => 'true'
]
```

- Description

```
[
    'attr' => ['class' => 'form-control'],
    'label' => 'Description',
    'label_attr' => ['class'=> 'form-label mt-4'],
    'required' => false,
]
```

- Submit :

```
[
    'attr' => ['class' => "btn btn-primary mt-4"],
    'label' => 'Ajouter un livre'
]
```

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

Template

Modification

Le principe ici est le même que pour la création.

Étant 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 fonction dans le controller.

La fonction prendra en plus en paramètre un livre qui sera passé directement à la fonction createForm.

Suppression

Définissez dans le controller une fontion `delete` qui prend en paramètre un `EntityManagerInterface` ainsi qu'un livre, et qui renvoie une objet de type `response`.

Flash message

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

Ajoutez dans le controller l'appel de fonction suivant lorsque c'est nécessaire :

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

Voici le code pour afficher les messages :

```
{% for message in app.flashes('error') %}
    <div class="alert alert-dismissible alert-danger mt-4">
        {{ message }}
    </div>
{% endfor %}
{% for message in app.flashes('success') %}
    <div class="alert alert-dismissible alert-success mt-4">
        {{ message }}
    </div>
{% endfor %}
```

Partie 3 : Les utilisateurs et les relations entre entités

À présent, nous allons gérer les comptes utilisateurs.

Comme expliqué dans le cours, un utilisateur est une entité particulière permettant de sécuriser une application Symfony. Commencez par faire un `make:user` pour créer l'entité. La propriété unique qui nous intéresse ici est le `username`, la classe doit donc porter l'attribut `#[UniqueEntity('username')]`.

Comme pour les livres, nous allons ajouter des contraintes sur les propriétés :

- le pseudo ne doit pas être vide, dépasser 30 caractères et en avoir moins de 3.
- le mot de passe ne pas être vide et doit faire au moins 8 caractères.

Une fois l'entité créée, il n'y pas grand chose à faire car Symfony s'est déjà chargé d'ajouter le provider pour (re)charger les données des utilisateurs dans la base de donnée ainsi que la fonction de mot de passe.

Inscription

Maintenant, nous pouvons gérer l'inscription des utilisateurs.

Créez un formulaire d'inscription à l'aide de la commande : `make:registration-form`. La commande crée à la fois un formulaire, un controller et un template.

Dans `RegistrationFormType`, supprimez la la ligne concernant le rôle car il ne servira pas pour le formulaire. Comme pour les livres, faites les modifications nécessaires pour la validation des données.

> Ici pour le mot de passe, nous allons plutôt utiliser un RepeatedType.

Symfony a quasiment tout fait ! Il ne nous reste plus qu'à tester !

Connexion

Cette fois-ci, nous allons essayer d'implémenter nous-même.

Dans `SecurityController`, ajoutez une fonction login qui pour l'instant affiche le template de connexion.

Plus tard, on voudrait que les utilisateurs non authentifiés soient redirigés vers la page de connexion lorsqu'ils tenteront d'accéder à une ressource sécurisée. Pour cela, rendez-vous dans `config > packages > security.yaml` et ajoutez le paramètre `form_login` en remplaçant route_name par le nom de la route créé précédemment :

```
security:
    # ...

    firewalls:
        main:
            # ...
            form_login:
                login_path: route_name
                check_path: route_name
                enable_csrf: true
```

Ensuite, retournez dans le controller pour modifier la fonction login. On va lui passer en paramètre un `AuthenticationUtils` qui va se charger de récupérer les erreurs en cas d'échec de la connexion. Il n'y a rien d'autre à faire car `FormLoginAuthenticator` se charge du reste.

Enfin, créez un template `login.html.twig` dans lequel vous rajouterez le code suivant :

```
{% if error %}
    <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
{% endif %}

<form action="{{ path('security.login') }}" method="post">
    <div class="form-group">
        <label for="username" class="form-label mt-4">Pseudo</label>
        <input type="text" class="form-control" id="username" name="_username">
    </div>
    <div class="form-group">
        <label for="password" class="form-label mt-4">Mot de passe</label>
        <input type="password" class="form-control" id="password" name="_password">
    </div>

    <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">

    <button type="submit" class="btn btn-primary mt-4">Se connecter</button>
</form>
```

À présent, il possible de créer un compte et de se connecter !

Déconnexion

On va rajouter le paramètre `logout: path: route_logout` dans le fichier `security.yaml` juste après `form_login`. Ensuite, on va ajouter une fonction `logout` qui ne fait et ne renvoie, car une fois de plus, Symfony fait tout ! ✨

Relations entre entités

Le but de cette partie est de permettre à un utilisateur d'emprunter des livres et de rendre un utilisateur propriétaire d'un ou de plusieurs.

Nous allons donc apporter des modifications à nos entités.

D'une part, on veut à la fois qu'un utilisateur soit propriétaire de 0 ou plusieurs livres et qu'il puisse emprunter 0 ou plusieurs livres. D'autre part, un livre ne peut avoir qu'un seul propriétaire et n'être emprunté que par un seul utilisateur. On aura donc :

- Dans User : deux relations `OneToMany` pointant vers `Book` ;

- Dans Book : deux relations `ManyToOne` pointant vers `User`.

Modifiez uniquement l'entité User à l'aide de la commande `make:entity`.

> Lorsqu'on ajoute une relation bidirectionnelle, modifié une seule entité suffit car dans le paramétrage dans le console fait automatiquement le lien pour la relation dans l'autre sens.