TP2: Base de données

Objectif

L'objectif de ce TP est de connecter une base de données à l'application et de créer une première entité.

Création d'une base de données

Création d'un container Docker

Dans un premier temps vous devez créer un container qui va contenir un serveur MySQL, il existe une commande pour créer un fichier de configuration Docker.

⚠️ Attention: s'il y a déjà un fichier *compose.yaml*, supprimez-le.

```
symfony console make:docker:database
```

Choisissez une base MySQL et laissez la valeur par défaut pour la version (latest).

```
- Docker Compose Setup-
-----------------------

Existing compose.yaml not found: a new one will be generated!

Which database service will you be creating?:
 [0] MySQL
 [1] MariaDB
 [2] Postgres
> 0

For a list of supported versions, check out https://hub.docker.com/_/mysql

What version would you like to use? [latest]:
>


created: compose.yaml

Success!

The new "database" service is now ready!

Next:
 A) Run docker-compose up -d database to start your database container
    or docker-compose up -d to start all of them.

 B) If you are using the Symfony Binary, it will detect the new service automatically.
    Run symfony var:export --multiline to see the environment variables the binary is exposing.
    These will override any values you have in your .env files.

 C) Run docker-compose stop will stop all the containers in compose.yaml.
    docker-compose down will stop and destroy the containers.
```

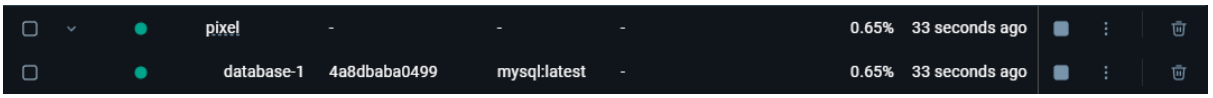
Pour modifier le nom de la table et le mot de passe, c'est dans le fichier *compose.yaml* que ça se passe !

Lancez la commande indiquée

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

Cette commande indique à Docker de monter le container "database" contenant le serveur MySQL.

Sur votre Docker Desktop vous devez voir le container:



Connexion à l'application Symfony

Les identifiants de la DB se fait dans le fichier *.env*

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=8.0.37"
```

Connexion à l'extension VS Code: Database Client

DB sans Docker

Si vous avez des difficultés pour faire fonctionner Docker, vous pouvez utiliser de SQLite par exemple, vous avez juste à modifier le fichier .env:

```
DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"
```

SQLite va stocker la base de données directement dans le dossier "data" du projet.
Lancez la commande suivante pour créer la db

```
symfony console doctrine:database:create
```

Première entité

Comme vu dans le cours, une entité est une classe PHP dont chaque instance va correspondre à un jeu de données (une ligne de la table).

```
symfony console make:entity Coaster
```

⚠️Respectez bien la case (minuscule au début et majuscule pour chaque mots).

Suivez ensuite les étapes pour créer une entité avec la attributs suivants:

name	string(80)	not null
maxSpeed	integer	null
length	integer	null
maxHeight	integer	null
operating	boolean	not null

Une fois la commande terminée, vous pouvez remarquer la présence d'un fichier *src/Entity/Coaster.php* et *src/Repository/CoasterRepository.php*

Migration

La création d'une nouvelle entité amène à modifier la structure de la base de données, il est alors nécessaire de créer une migration, là encore une commande est là !

```
symfony console make:migration
```

Exécutez ensuite la migration.

```
symfony console doctrine:migrations:migrate
```

Vous pouvez maintenant vérifier que la table à bien été créée dans VS Code.

Nous allons ensuite créer le CRUD de l'entité Coaster, le CRUD est l'acronyme de **Create-Read-Update-Delete.**

Ajouter un nouveau Coaster

Pour le moment nous n'avons pas encore vu comment créer un formulaire, nous allons donc créer un coaster à la volée.

Coaster Controller

Créez un fichier *CoasterController.php* dans le dossier Controller du projet, reportez vous sur le fichier AppController déjà présent.

Pour ajouter une entité en base de données, vous aurez besoin d'un objet de type *EntityManagerInterface*. Pour l'obtenir dans le controller… demandez le !

Grâce un design pattern intégré dans Symfony: l'**injection de dépendances (DI)**

```php
public function add(EntityManagerInterface $em): Response
{
    $entity = new Coaster();
    $entity->setName('Blue Fire');
    // ...

    $em->persist($entity); // Ajoute l'entité dans le manager
    $em->flush(); // Exécute les requêtes

    return new Response('Coaster ajouté');
}
```

⚠️ **N'oubliez pas d'ajouter la route**

Si vous chargez le lien sur votre navigateur, une entrée dans la base de données devrait apparaître.
Nous faisons appel à l'objet EntityManager qui va permettre d'insérer une nouvelle entité.

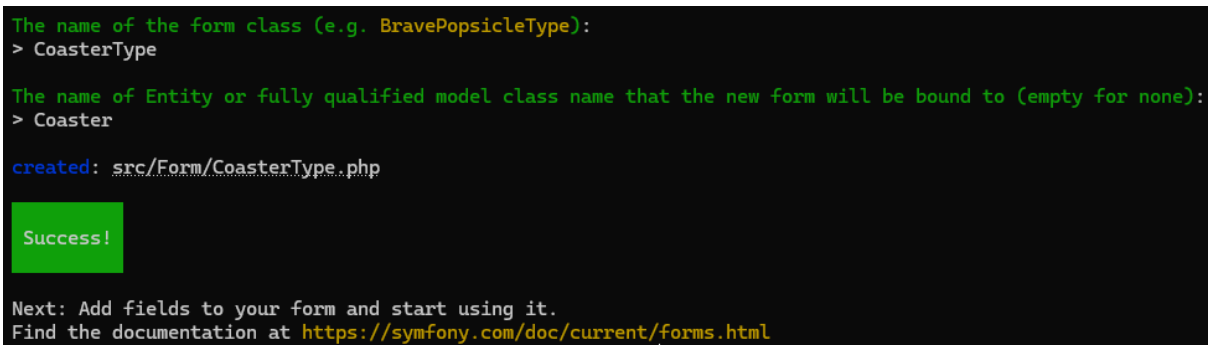
Avec un formulaire c'est mieux !

On ajoute bien un Coaster mais ce n'est pas très pratique, nous allons donc faire un vrai formulaire avec une page ergonomique.

La gestion d'un formulaire se fait dans une classe à part, pour cela il y a encore une commande:

```
symfony console make:form
```

Définir "CoasterType" pour le nom de la classe.



Ouvrez le fichier créé.

```php
class CoasterType extends AbstractType
{
    public function buildForm(
        FormBuilderInterface $builder,
        array $options
    ): void
    {
        $builder
            ->add('name')
            ->add('maxSpeed')
            ->add('length')
            ->add('maxHeight')
            ->add('operating')
        ;
    }

    public function configureOptions(
        OptionsResolver $resolver
    ): void
    {
        $resolver->setDefaults([
            'data_class' => Coaster::class,
        ]);
    }
}
```

La méthode “buildForm” va définir les champs de formulaire à afficher. “configureOptions” définies par défaut l’entité Coaster comme donnée de référence, c'est-à-dire que le formulaire va se baser sur cette classe pour créer les types de données et donc les types de champ.

Modifier la méthode du controller pour ne plus créer automatiquement les données mais afficher un formulaire:

```php
public function add(EntityManagerInterface $em): Response
{
    $entity = new Coaster();
    $form = $this->createForm(CoasterType::class, $entity);

    return $this->render('coaster/add.html.twig', [
        'coasterForm' => $form,
    ]);
}
```

Pour afficher un formulaire dans une vue Twig utilisez la ligne suivante:

```twig
{{ form_start(coasterForm) }}

{{ form_widget(coasterForm) }}

{# Bouton HTML submit #}

{{ form_end(coasterForm) }}
```

Traitement du formulaire

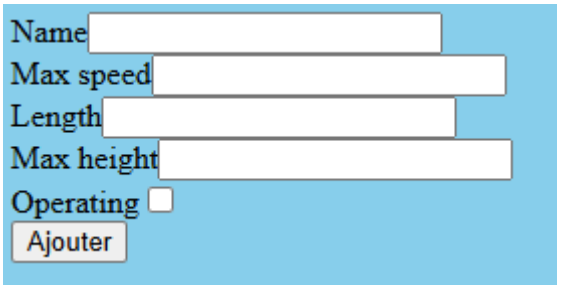
Avant d'envoyer les données du formulaire en base de données il faut s'assurer de 3 choses:

- Récupérer les données envoyées
- Tester que le formulaire est bien envoyé
- Tester la validation des données

Dans un formulaire, la méthode "handleRequest(Request $request)" permet de récupérer les données POST et de remplir les champs du formulaire.


```php
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Component\HttpFoundation\Request;

public function add(
    Request $request,
    EntityManagerInterface $em
): Response
{
    // ...

    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $em->persist($entity);
        $em->flush();

        return $this->redirectToRoute('app_app_index');
    }

    // ...
}
```

Et voilà votre formulaire opérationnel:



Un peu de design 🎨

Thème DaisyUI pour les formulaires

Dans un premier temps nous pouvons remarquer que le formulaire ne prend pas en compte DaisyUI, c'est normal car par défaut Symfony n'ajoute pas les classes css. Nous allons donc indiquer dans la configuration qu'il faut utiliser le template de formulaire pour DaisyUI.

Téléchargez tout d'abord le template à cette adresse: https://raw.githubusercontent.com/symfony/symfony/f8182b17892fcc24547ae1691c82cf853ab2322f/src/Symfony/Bridge/Twig/Resources/views/Form/daisyui_5_layout.html.twig

Copier ce fichier dans le dossier *templates/form* du projet.

Rendez-vous sur le fichier *config/packages/twig.yaml* et ajoutez la ligne suivante juste en dessous de la ligne 2:

```
form_themes: ['form/daisyui_5_layout.html.twig']
```

⚠️Le yaml fonctionne avec l'indentation, il faut bien ajouter cette ligne avec la même indentation que la ligne 2 (*file_name_pattern*).

Modifier le layout

Autre problème, il n'y a aucune marge, modifier le fichier *templates/base.html.twig*

```
<body>
    <main class="flex flex-nowrap h-[100vh]">
        <div class="flex flex-col shrink-0 shadow w-[240px]
bg-base-200">
            <div class="py-2 bg-primary text-center text-white">Wiki
Coaster</div>
        </div>
        <div class="p-4 flex-grow-1">
            <div class="bg-base-200 p-2 rounded-xs border-base-300">
                {% block body %}{% endblock %}
            </div>
        </div>
    </main>
</body>
```

N’oubliez pas la commande pour construire les assets:

```
npm run watch
```

Vous devriez avoir un rendu plus agréable.

Afficher la liste des coasters

Maintenant que l'on peut ajouter des coaster, vous pouvez maintenant créer une nouvelle page pour afficher les la liste (et administrer).

Ajoutez une méthode *index* dans le *CoasterController*, pour obtenir la liste des coasters, appelez la méthode ***findAll*** de l'objet ***CoasterRepository*** (en utilisant l'injection de dépendances).

N'oubliez pas d'envoyer la liste dans la vue pour y avoir accès.

Bouton d'ajout

Depuis cette liste, ajoutez un lien qui va diriger vers le formulaire d'ajout, dans l'attribut *href* de la balise html, utiliser la fonction Twig suivante:

```
{{ path('app_coaster_add') }}
```

Note: cette fonction demande le nom de la route, si vous ne savez pas le nom la commande suivante affiche toutes les routes disponibles:

```
symfony console debug:router
```

Utilisez DaisyUI pour afficher le lien comme un bouton.

Modifier un coaster

La modification est très similaire à l'ajout, la différence est que l'entité n'est pas créée mais récupérée depuis la DB.

Ajouter un paramètre dans une route

Ajoutez un paramètre *id* dans la route en vous aidant de la documentation: https://symfony.com/doc/current/routing.html#route-parameters

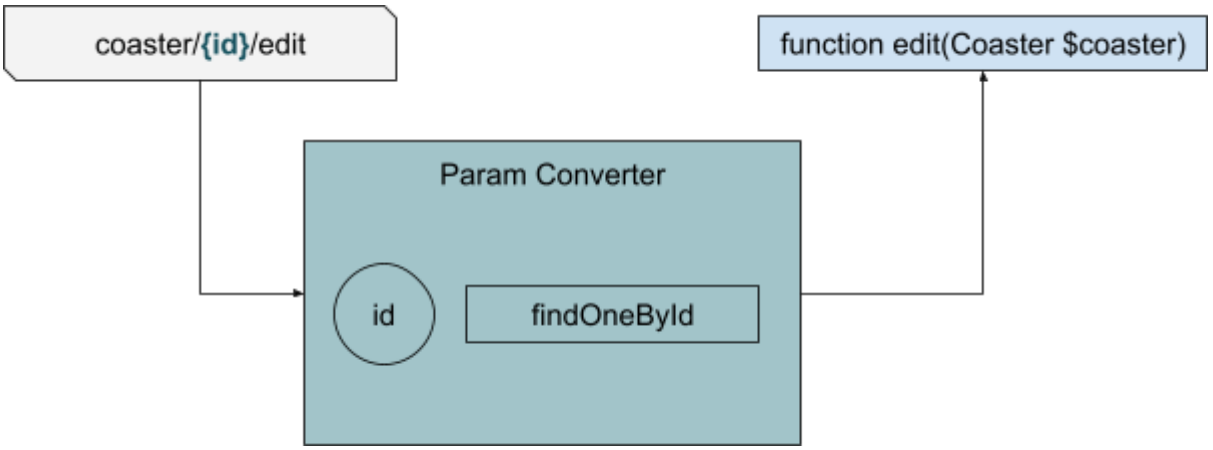
Grâce à l'id il est donc possible de faire une requête à la DB via Doctrine. Il existe une fonctionnalité dans Symfony qui permet de convertir automatiquement un paramètre en entité, ça s'appelle le ***Param Converter***.

Celui-ci va essayer de trouver l'entité en fonction du typage, ainsi si vous ajoutez la paramètre suivant dans la méthode:

```
public function edit(Coaster $coaster)
```

Note: bien sûr il ne faut pas que le paramètre $coaster

Symfony va faire le lien entre le nom du paramètre "*id*" et le type d'objet "*Coaster*" pour faire automatiquement la requête.



Le reste du code reste le même à un détail près: comme l'entité est déjà chargée, elle est déjà gérée par Doctrine, il n'est pas nécessaire d'appeler la méthode "*persist*".

Bouton "Modifier"

Ajoutez maintenant un lien dans la page de la liste pour modifier chaque coaster, aidez vous de la doc pour ajouter la route et l'id: https://symfony.com/doc/current/reference/twig_reference.html#path

Supprimer 🗑️

Il ne reste plus qu'à supprimer, cette action est critique, pour éviter que l'utilisateur supprime par inadvertance un coaster nous allons passer par un message de confirmation.

Écrivez d'abord une méthode "*delete*" avec un paramètre de la même manière que l'édition.

Protection CSRF

Nous allons ajouter une protection CSRF pour la suppression avec un formulaire qui sera composé seulement du *token* et du bouton *submit*.

```
<form method="post">
    <h1>Supprimer {{ coaster.name }}</h1>

    <p>Êtes-vous sûr de vouloir supprimer ce coaster ?</p>

    <input type="hidden" name="_token" value="{{ csrf_token('delete'
~ coaster.id) }}">
    <button type="submit" class="btn btn-warning">Supprimer</button>
</form>
```

Ajoutez cette condition pour tester si le token est valide:

```php
if ($this->isCsrfTokenValid(
    'delete'.$coaster->getId(),
    $request->request->get('_token')
)) {
    $em->remove($coaster);
    $em->flush();

    return $this->redirectToRoute('app_coaster_index');
}
```

Il vous reste à ajouter le bouton dans la liste.

Notre CRUD est terminé 🎉