ÉNONCE – GESTIONNAIRE D'INVENTAIRE AVEC SYMFONY

Objectif :

L'objectif est d'améliorer la prise en main de **Symfony**. Avec pour exemple un outil d'inventaire informatique, vous partirez d'un squelette de projet que vous complèterez par diverses fonctions. Ainsi, pour réaliser ces fonctionnalités, vous manipulerez les étapes clés dans **Symfony** pour la réalisation de sites complexes. Vous travaillerez dans un premier temps avec **Doctrine**, qui est l'**ORM** intégré à **Symfony**, pour la manipulation des données. Puis nous travaillerons sur l'élaboration de formulaires et proposer les 4 actions de bases : Création, Lecture, Mise à jour, Suppression. Ces actions sont appelées **CRUD** en anglais dans les documentations.

Ressources indispensables :

- https://www.php.net/docs.php
- https://laconsole.dev/formations/symfony

Etape 1 : On s'installe

Un dépôt du projet est à récupérer sur Github. Vous pouvez récupérer le squelette du code avec la commande **git clone https://github.com/laurentgiustignano/R3.01-GestionInventaire-Enonce** dans votre répertoire de travail. Ou bien depuis **PhpStorm**, en choisissant : **Get from Version Control** avec la même URL.

Le projet comporte 4 éléments :

- **EquipementController.php** qui est le contrôleur les accès à **Equipement**. Il contient pour l'instant une route en **/equipement**.
- **templates/base.html.twig** et **templates/equipement/index.html.twig** qui sont les deux fichiers qui sont utilisés pour le rendu des vues.
- **.env** qui contient la description pour la connexion à votre serveur de base de données. Il est à modifier en premier lieu.

Le projet que vous avez cloné correspond à celui qui fonctionnait sur ma machine, et donc avec les informations d'identification pour mon serveur de base de données, et son nom de base de données. Vous devez faire correspondre dans ce fichier les informations concernant votre serveur, ainsi qu'un autre nom, si vous choisissez d'en changer. Il n'est pas non plus important que la base soit déjà présente sur le serveur, car nous verrons que Symfony se chargera de la créer.

- Dans le fichier **.env**, modifiez l'attribut **DATABASE_URL** en commentant/décommentant la ligne pour qu'elle correspondent au information relative à votre serveur de base de données (MySQL, MariaDB, PostgreSQL ou SQLite). Sur ma ligne active, le terme 'root' à gauche du symbole ' :' correspond au nom de l'utilisateur, l'autre au mot de passe, et enfin le terme 'inventory' correspond au nom de la base sur le serveur.
- Comme les informations de connexion sont enregistrées, Symfony va pouvoir accéder au SGBD. Dans un terminal au niveau de votre projet, saisir la commande : **php bin/console doctrine:database:create**.

```
┌─laurent at MacBook Pro in ~/Documents/Scolaire/ProjetsPhpStorm/inventory
└─.:: php bin/console doctrine:database:create
Created database `inventory` for connection named default
```

Figure 1- Exemple de création de base de données réussi

Etape 2 – « Space Entity »

Pour créer une table qui sera en relation avec Symfony, nous allons créer une **entity**. Il s'agit qu'une classe PHP qui fera l'interface entre les données enregistrées dans le SGBD et la logique de votre application, qui sera dans les contrôleurs. L'outil **make:entity** de **Symfony** permet au développeur de décrire la structure des données, et en retour va générer un fichier d'entité **PHP**, dans le dossier **src/Entity**, contenant tous les getters et setters permettant à un objet, de ce type d'entité, de représenter les données. Ensuite, avec l'outil **make:migration**, **Symphony** va créer générer un fichier permettant de travailler avec une version de BDD définie. Ainsi, il est plus facile de travailler à plusieurs sur un projet et surtout avec la même base de données. Pour terminer et effectuer les changements dans la BDD, la commande **doctrine:migrations:migrate** est nécessaire. Après confirmation, l'écriture se fera et vous pourrez vérifier via un client SQL.

- Dans un terminal au niveau de votre projet, créer l'entité **equipement**. Les champs à créer sont : **titre**, **description**, **categorie**, **stock**. Hormis les trois premiers qui sont toutes de type **string**, le dernier champ sera un **integer**.
- Parcourrez le fichier créé **src/Entity/Equipement.php** pour consulter les différents éléments de la classe. Questionnez-vous sur l'intérêt du **return $this** pour les **setters** ?
- Générer le fichier de migration en tapant la commande : **php bin/console make :migrantion**. Observez le contenu du fichier généré. Si le contenu SQL correspond bien aux définitions établies, vous pouvez passer à l'étape suivante.
- Effectuer l'enregistrement dans la base de données avec la dernière commande.

Remarque : Vous pouvez utiliser l'utilitaire intégré à PhpStorm pour accéder à votre base de données. En général, vous le trouvez sous la forme d'une icône « burger » dans la partie droite de votre fenêtre. Sinon, vous pouvez y accéder depuis le menu **View/Tool Windows/Database**. *Ajoutez une nouvelle source sur le bouton* **+**, *puis vous n'avez qu'à saisir les mêmes informations que dans votre fichier* **.env**. *Vous pouvez tester la connexion avec votre base avant de valider. Une fois fait, vous pouvez naviguer dans votre serveur et vérifier le contenu de votre base* **equipement**.

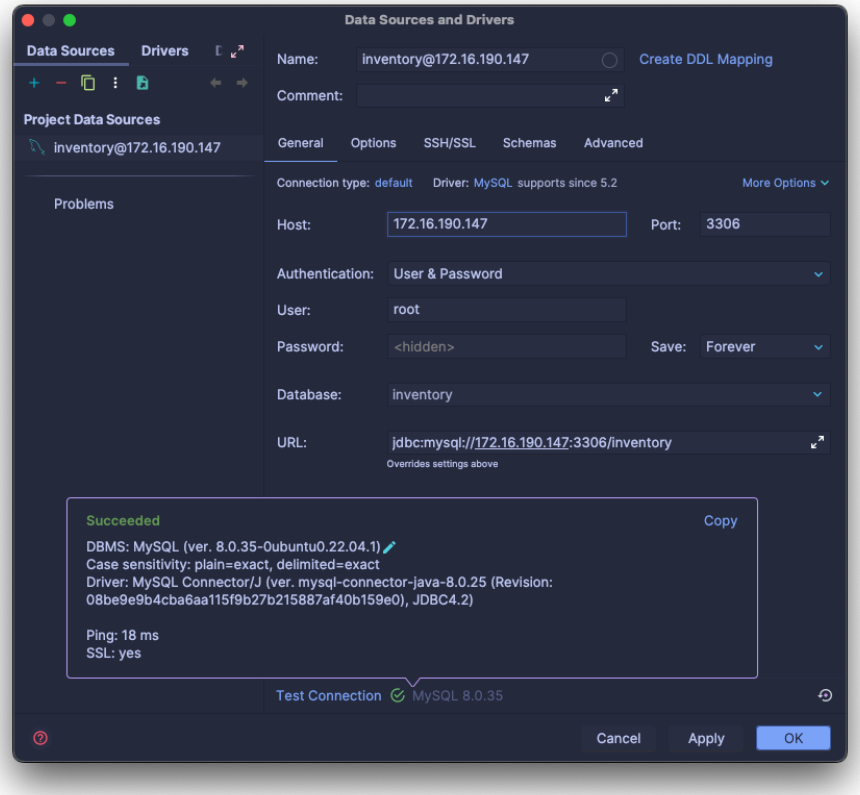

Figure 2- Exemple de configuration de l'outil Database de PhpStorm

Étape 3 – Y'a quelqu'un ?

Maintenant que vous avez une base de données, il est temps de regarder ce que le site propose. Le contrôleur **EquipementController** possède une route qui s'appelle **equipement_index** et qui répond sur la méthode **GET** depuis l'url **/equipement**. La fonction **index()**, qui est exécuté lorsque cette route « est prise », possède un paramètre. Il est de type **EquipementRepository** correspond à une des classes qui a été générée lorsque l'entité **Equipement** a été créée. Elle permet de récupérer les informations depuis la base via les méthodes **find()**, **findAll()**, **findOneBy()** et **findBy()**. Ainsi, on peut récupérer les informations contenues dans la base et les envoyer au moteur de template : **twig**. Une page de template spécifique est définie dans le fichier **templates/equipement/index.html.twig** qui reçois, dans un tableau associatif, les valeurs que nous souhaitons utiliser. Ici, il s'agit des données issue de la table **Equipement** que nous récupérons avec l'appel de **$equipementRepository->findAll()**.

- Dans un terminal au niveau de votre projet, démarrer le serveur interne de **Symfony** avec la commande : **symfony server:start -d**. Observez l'affichage et déduire le comportement de la structure **{% for in %} … {% else %} … {% endfor %}**. Vous noterez aussi que l'inventaire est pour l'instant vide.
- Ajoutez un équipement avec le client SQL de votre choix, puis rechargez la page.

Remarque : Nous conviendrons que l'insertion de données via le client n'a qu'un but pédagogique pour illustrer le fonctionnement initial du site.

Voyons comment insérer des données depuis **Symfony**. Pour se faire, nous créer une nouvelle route dans **EquipementController** en dupliquant la méthode **index()**. Nous appellerons cette méthode **ajout()**, et nous souhaitons, qu'en plus d'afficher les éléments du repository, elle puisse insérer des données dans la table. Pour cela, elle doit travailler avec l'entité. **Symfony** propose une interface, par le biais de la classe **EntityManagerInterface**, pour communiquer la BDD. En créant un objet **Equipement**, en lui donnant des valeurs, l'interface pourra rendre ces données persistantes en les enregistrant dans la table. Comme il y a déjà une relation entre les attributs de l'objet de classe **Equipement** et les champs de la table **Equipement**, il n'y a rien d'autre à faire que de synchroniser les données avec la méthode **flush()**.

- Dupliquer le code de la méthode **index()** pour créer une nouvelle méthode **ajout()**. L'attribut de la nouvelle route correspondante sera : **#[Route('/ajout', name: 'equipement_ajout', methods: ['GET'])].** Ajouter un paramètre à la méthode de type **EntityManagerInterface**, usuellement appelé **em**, pour **Entity Manager**.
- Dans la méthode, vous devez créer un objet de type **Equipement**. Puis avec les **setters**, donnez des valeurs à cet objet. N'oubliez pas que les **setters** sont **fluents**.
- Enfin, l'objet **$em** doit utiliser la méthode **persist()** avec l'objet **Equipement** comme paramètre, puis faire la synchronisation.
- En chargeant la page sur l'url **/ajout**, vous devriez observer la présence de votre nouvel équipement enregistré dans la BDD.

Étape 4 – On ne voit que lui

Tous les équipements sont visibles, mais il serait intéressant de visualiser un seul élément. On pourrait imaginer par la suite avoir une image qui correspond à l'équipement mais son affichage dans le tableau récapitulatif n'a pas d'intérêt. Il faut donc être capable de « voir » un seul équipement. Par contre, il doit être possible de sélectionner un équipement. En parcourant le fichier **Equipement.php**, vous avez remarqué que **Symfony** a rajouté une propriété **$id** qui est clé primaire de la table. **Symfony** nous permet de créer une route '**/equipement/{id}**' avec une méthode qui reçoit un objet de type **Equipement**, cela signifie que cet objet aura été mappé avec les données de la table en fonction de l'id

fourni dans l'url. Nous pouvons créer une nouvelle page **show.html.twig** qui affichera, dans une **card** de **bootstrap**, les informations d'un équipement, comme ci-dessous :

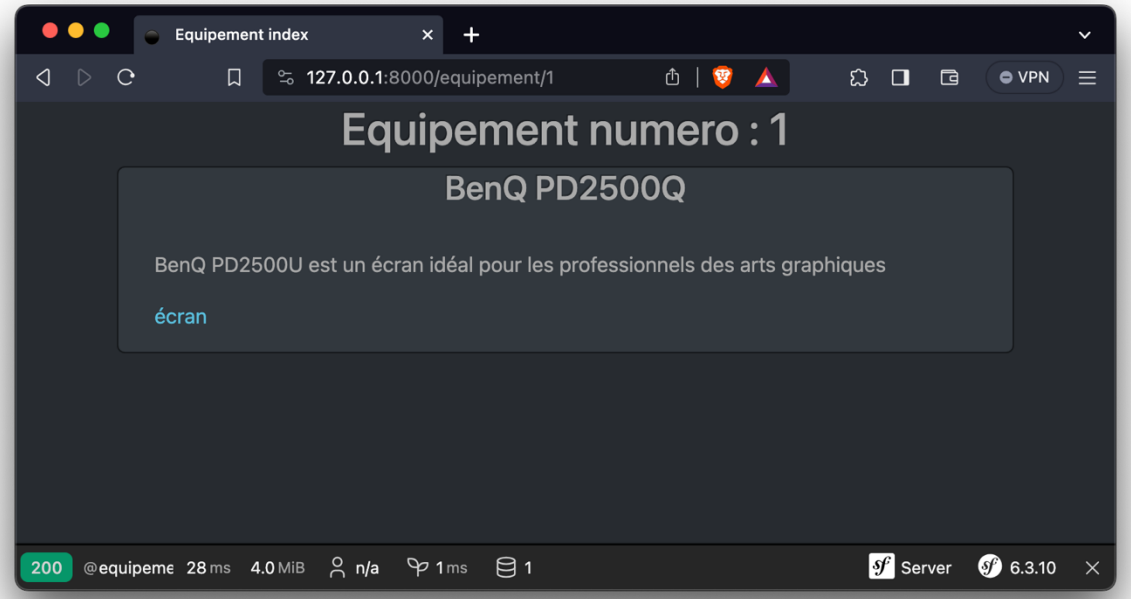

Figure 3 - Exemple d'affichage pour un équipement

- Créer la route demandée, la méthode **show()** et la page **twig** correspondante.
- Nous allons utiliser cette nouvelle route pour ajouter une fonctionnalité à la page **index.html.twig**. Créer un lien pour chaque équipement pour qu'il amène à la page **show.html.twig**. Pour cela, il faut ajouter une balise lien dont le **href** sera égal à : **{{ path('nom_de_la_route', {id: equipement.id}) }}**.

Étape 5 – Presque pareil !

Pour terminer, nous allons améliorer le site pour pouvoir modifier des valeurs par le biais d'un formulaire. **Symfony** propose un outil pour simplifier leur création. Avec **make:form**, on va générer la classe décrivant notre formulaire. Par convention, son nom se termine par Type : nous l'appellerons **EquipementType**. Ensuite, il faut spécifier le nom de l'entité qui sera en lien avec le formulaire. Pour ce projet, il s'agit de l'entité **Equipement**. Avec ces deux informations, le fichier **src/Form/EquipementType.php** contiendra la description du formulaire. En parallèle, il faut créer dans le contrôleur une nouvelle route, une nouvelle méthode et une nouvelle page **twig** pour effectuer la modification d'un équipement. Je propose de les appeler '**/equipement/{id}/edit**' , **edit()** et **edit.html.twig**.

Dans la méthode **edit()**, pour utiliser le formulaire, on peut faire appel à la méthode **createForm()** qui reçoit en premier paramètre la classe Type du formulaire, et en deuxième paramètre les données à insérer dans les champs qui peut-être un tableau, ou bien directement une entité. Cette fonction retourne un objet de type **FormInterface**, représentant le formulaire qui sera utilisé dans **twig**.

- Créer la route demandée, la méthode **edit()** et la page **twig** correspondante. Il faut s'inspirer du fichier **show.html.twig** et même reprendre le début de la **card**.
- Pour représenter l'ensemble du formulaire, il faut utiliser dans l'ordre ces trois **helper** de **twig** :
  - **form_start()** : pour commencer le formulaire avec comme paramètre l'objet **FormInterface** créé dans le contrôleur.
  - **form_widget()** : pour présenter tous les champs du formulaire. Il aura le même objet en paramètre.
  - **form_end()** : pour terminer le formulaire. Également avec le paramètre.

- En affichant la page '**/equipement/1/edit**', vous observez la page avec le formulaire qui contient les données du premier équipement. Ajouter un bouton pour enregistrer le formulaire. En rechargeant la page, vous tomberez sur une erreur quand vous cliquerez sur le bouton. En effet, fournir des données s'effectue avec la méthode **POST**. Il faut rajouter **POST** aux méthodes de la route.
- Dans la méthode **edit()**, il faut récupérer les éléments transmis. On a besoin d'un objet de type **Request** qui représente la requête. En le rajoutant en paramètre, il est automatiquement injecté par **Symfony**. De plus, comme des modifications vers la BDD vont être effectuées, il faut à nouveau utiliser l'objet **EntityManagerInterface**. Ainsi, il ne reste qu'à « pousser » les données vers la base, puis rediriger l'utilisateur vers la route choisi, par exemple **equipement_index**. Insérer le code suivant entre la création de l'objet **$form** et le **return** :

```php
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid()) {
    $entityManager->flush();

    return $this->redirectToRoute( route: 'equipement_index');
}
```

Pour aller plus loin…

Même si on est pas mal loin, quelques pistes pour poursuite :

- Par exemple, on pourrait rajouter un bouton **éditer** sur l'index des équipements pour diriger sur la route **equipement_edit**
- Et aussi un bouton **delete**, pour supprimer un équipement. Mais il faut faire la route, la méthode, etc… ;)

Bon courage…