Projet de Web - Yann Berthelot

Table des matières

Erratum

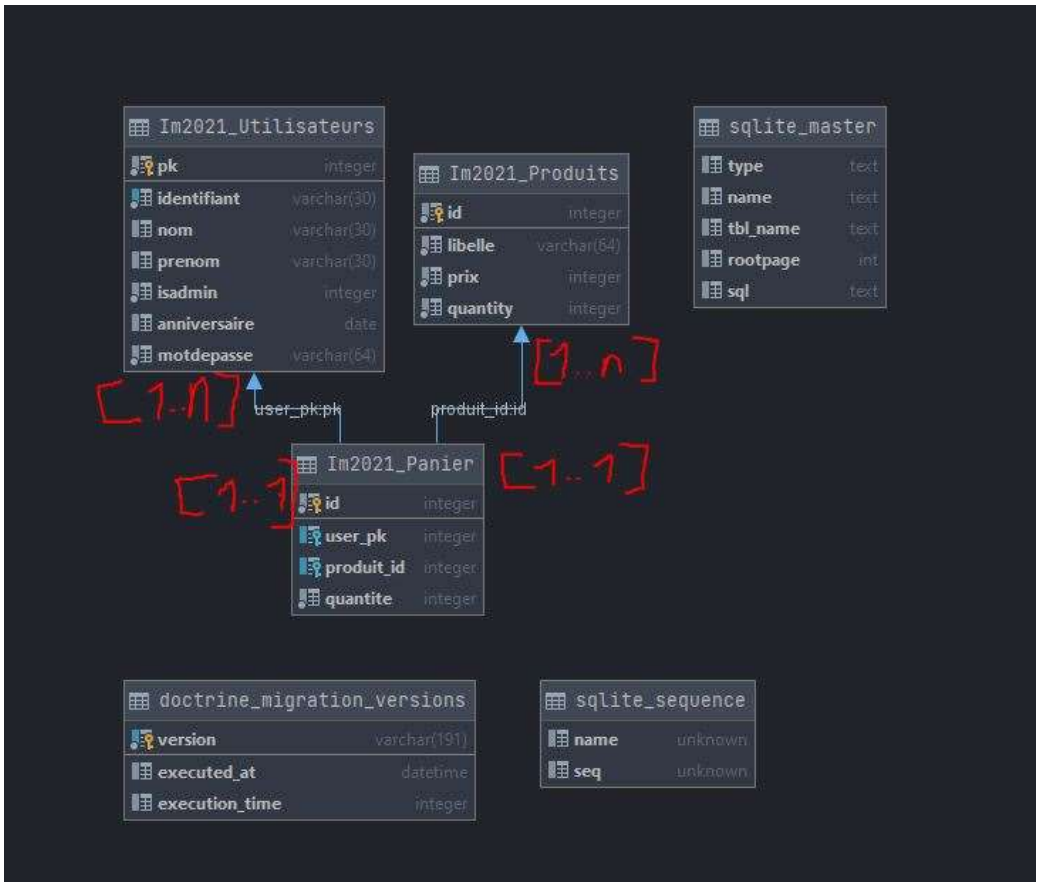
Je commence ce rapport en m'excusant pour l'absence de binôme sur ce groupe, en effet, avec mon déménagement à Bordeaux et l'organisation de mon stage en Italie dans un timing très serré, j'ai commencé le projet tard (14/04) et l'ai fait le plus rapidement possible sans attendre une réponse tardive des personnes auxquelles j'ai proposé de se joindre à moi. Je m'attend à être sanctionné et le but de cette explication n'est pas de l'éviter, je souhaite juste expliquer la situation car je pense qu'il est toujours préférable de se justifier afin de montrer que ce n'est pas simplement sur un coup de tête que j'ai fait ce projet en monôme malgré la consigne.

Contexte

Le but de ce projet est de créer un site web de vente permettant de manipuler une base de données sans gestion d'authentification pure. Un paramètre global "id" dans le fichier `config/services.yaml` qui référence la clé primaire de la table utilisateurs sert à la vérification du statut de l'utilisateur connecté.

Organisation interne

Le schéma de la base de donné est le suivant:



Nous avons 2 relations entre l'utilisateur et un panier et un produit eet un panier. Le panier est un peu mal nommé, en effet un `panier` est plus exactement une commande, une ligne de la table référence un utilisateur, un produit et contient la quantité souhaitée. Un client commandant 3 articles générera donc 3 lignes dans la table `Panier`. L'avantage des relations mises en place est de pouvoir accéder à n'importe quel champ du produit ou et du client référencé dans la commande, ce qui simplifie grandement la gestion et l'affichage du panier notamment.

La hiérarchie du code, elle, est la suivante :

```
.
├── composer.json
├── composer.lock
```

```
├── public
│   ├── css
│   │   ├── accueil.css
│   │   ├── base.css
│   │   └── layout.css
│   ├── Images
│   │   ├── admin_banner.jpg
│   │   ├── footer.jpg
│   │   ├── guest_banner.jpg
│   │   └── user_banner.jpg
│   └── index.php
├── sqlite
│   └── data.db
├── src
│   ├── Controller
│   │   └── ProjetController.php
│   ├── DataFixtures
│   │   └── AppFixtures.php
│   ├── Entity // les tables de la BDD
│   │   ├── Panier.php
│   │   ├── Produit.php
│   │   └── Utilisateurs.php
│   ├── Form //les formulaires de creation de produit et de compte utilisateur
│   │   ├── AjoutProduitType.php
│   │   └── CreationCompteType.php
│   ├── Kernel.php
│   ├── Repository
│   │   ├── PanierRepository.php
│   │   ├── ProduitRepository.php
│   │   └── UtilisateursRepository.php
│   └── Service // un dossier créé à la main tout comme le service qu'il contient
│       └── myService.php
├── symfony.lock
└── templates // les templates contenues dans 'projet' héritent de celui dans 'layouts'
qui hérite de base.html.twig
    ├── base.html.twig
    └── projet
        ├── Accueil.html.twig
        ├── AjoutProduit.html.twig
        ├── connexion.html.twig
        ├── creationCompte.html.twig
        ├── gestionPanier.html.twig
```

```
├── GestionUsers.html.twig
├── index.html.twig
├── Layouts
│   └── layout.html.twig
└── listProduits.html.twig
```

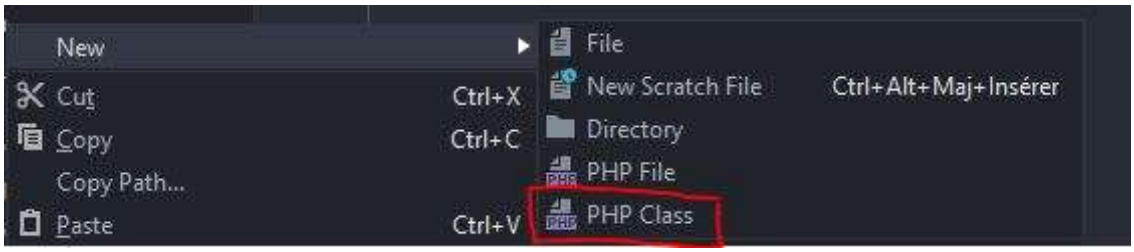
Les points importants sont les suivants :

- `service.yaml` : Le fichier contenant la variable globale `id` qui gère l'authentification
- `site/public` : Contient les images et feuilles de styles utilisées
- `site/sqlite` : Contient la base de donnée utilisée dans tout le site
- `site/src` : Le gros du site, contient le controller, les Tables, les formulaires, le service et les templates créés

Tuto Service

Creer son propre service symfony est plutôt simple :

1.Creer sa classe php, de préférence dans un répertoire dédié



2.On ajoute sa fonction dans le code généré, pour un service qui renvoi l'inverse d'un string on obtient :

```php
<?php


namespace App\Service;


class myService
{
    public function invertMyMessage(): string
    {
        $source = "Shrekos";
        return strrev($source);
    }
```

```
}
```

3.on l'inclut dans une action du controller en passant le service en argument :

```
use App\Service\myService
class ProjetController extends AbstractController
{
    /..
    public function accueilAction(myService $myservice): Response
    {
       //..
        $args = array
        (
            'status' => $status,
            'user' => $user,
            'serviceResult' => $myservice->invertMyMessage()
        );
        return $this->render('projet/accueil.html.twig', $args);
    }
    //..
```

Points particuliers et difficultés

J'ai pu témoigner de la rapidité et de la puissance du framework grâce à ce projet, en effet malgré le commencement tardif du projet, j'ai vite été rassuré en voyant qu'au bout de 5 heures le plus gros était fait malgré les difficultés rencontrées : j'ai dû recommencer 2 fois le projet à cause de mauvaises manipulations sur doctrine qui m'empêchaient de migrer de version que ce soit pour une version plus récente ou un retour en arrière. J'ai également rencontré un problème avec les commentaires des Tables, j'ai en effet utilisé l'option `Comment` mais lors de la génération de code SQL ils n'ont pas été retranscrits. J'ai lu que l'option n'était prise en compte que lors de la re-création de la table mais avec les difficultés précédentes je n'ai pas tenté le diable. Le seul point particulier que je peut évoquer est vraiment les relations entre entités qui furent un gain de temps considérable et ce fut très satisfaisant de s'en servir une fois bien mises en place.