SOUCHET Mathéo

MILLET Arthur

PRESTI Louis

Compte Rendu SAE Qualité de Développement

A/ Premier Design Pattern: Factory

1/ réponse à un problème.

Pour pouvoir insérer de nouvelles données dans la table, comme un produit, un user, un code promo, il nous est normalement nécessaire de devoir créer un instance de l'entité correspondante et ainsi créer notre objet avec les données voulues.

Cela nous permet dans un premier temps d'éviter de créer une instance pour rien, de simplifier la compréhension du code (séparation des préoccupations)

2/ Mise en œuvre.

L'implémentation de la Factory s'est révélée intuitive est non complexe.

Nous avons implémenté notre méthode statique dans les entités suivantes:

-CodePromo

-Panier

-Produit

Pour pouvoir l'implémenter dans notre Diagramme de classe niveau conception, nous avons donc rajouté une méthode statique (symbolisé par un soulignage) pour ces classes là.

3/ Développement PHP

La forme de l'implémentation est identique pour chacune des classes. Je prendrais ici l'exemple avec l'entité "produit" (méthode statique implémentée dans les entités) .

```
public static function factory(string $nom,float $prix,int $quantite,string
$description,string $categorie,string $souscategorie,string $image): self{

    $produit=newself();
    $produit->setNom($nom);
    $produit->setPrix($prix);
    $produit->setQuantite($quantite);
    $produit->setDescription($description);
    $produit->setCategorie($categorie);
    $produit->setSousCategorie($souscategorie);
    $produit->setImage($image);return $produit;
}
```

B/ Deuxième Design Pattern: MVC

1/ réponse à un problème.

Très fortement conseillé, le design Pattern MVC (Modèle, vue, Contrôleur) à une place centrale dans notre conception du site web.

En effet, du fait de nombre de fichiers conséquents, nous devons apporter une clarté supplémentaire à notre arborescence de fichier pour nos classes métiers.

Nous avons donc représenté Les modèles par nos entités (Comme vu à la question précédentes).

Nos contrôleurs seront représenté par les différentes classes situées dans notre dossier Contrôleur.

Enfin , étant donné que la vue est géré par un simple fichier ".twig", il est donc pas possible de le représenter dans le diagramme de classes. **C'est pourquoi nous avons décidé de considérer la vue et le contrôleur comme une seule entité dans le diagramme** (étant donné que c'est le contrôleur qui va aller chercher le fichier ".twig" et c'est ce dernier qui se charge de renvoyer la réponse à Symfony qui se chargera d'afficher la page en conséquence.

2/ Mise en œuvre:

Par rapport à notre diagramme de classe d'analyse, nous avons donc dû totalement revoir notre conception du site. De nombreuses classes ont donc maintenant été ajoutée. Vu que notre MVC s'applique à l'ensemble de notre Diagramme, voici une capture d'écran de notre diagramme UML.

Toutes les images sont disponibles dans le dossier img/

3/ Développement en PHP.

Notre architecture MVC se déploient de cette manière dans notre arborescence de fichier:

NB: La table Avis représentée ici d'existe plus et sera enlevé dans le futur.

Voici un exemple concret où un contrôleur appelle un fichier twig afin de pouvoir rendre une page:

```
class HomeController extends BaseController{

   #[Route('/', name: 'home.index')] // Géré par Symfony

   //Reponse est un type de variable géré par Symfony pour gérer l'affichage de la
page twig           //l'affichage de la page

        public function index(): Response{
            $this->addGlobalVariables();

            return $this->render('home/index.html.twig');
        }

}
```

C / Troisième Design Pattern: DAO

1/ réponse à un problème.

Pour que notre site web soit le plus réel possible, nous avons dû avoir besoin d'une base de donnée. Pour pouvoir lier notre base de donnée, le design pattern DAO nous a paru comme la meilleur solution à ce problème (En réalité, nous avions déjà anticipé ce problème dû à de nombreuses implémentations de base de données déjà faite dans d'autres projets).

2/ Mise en œuvre:

Tout d'abord nous avons créé des classes Repository, permettant de faire le parallèle direct avec notre base de données.

Ensuite cela pourra être récupéré grâce aux entités (équivalentes aux data class en Kotlin).

Voici Un exemple dans notre diagramme UML de la partie User concernée par le DAO:

Le Repository est utilisé par par le controlleur.

Et le Respository Utilise la classe User.

3/ Développement en PHP:

En complément du design Pattern MVC, Nous avons donc implémenté notre DAO de la manière suivante:

En effet dans elle y contient une méthode __construct() où va permettre une connexion à notre database sur mariadb

C'est pourquoi il sera nécessaire à chaque modification/suppression ou insert dans la base de donnée d'appeler le repository correspondant.

Voici un exemple (très) concret d'un repositiory ainsi qu'un exemple de son utilisation dans un controlleur:

```
class CodePromoRepository extends ServiceEntityRepository{
    private $connexion;

    public function __construct(ManagerRegistry $registry,Connection $connexion){
        parent::__construct($registry, CodePromo::class);
        $this->connexion = $connexion;
    }

    public function ModifierCode($id,$pourcentage,$nbr_use,$nom_code){
        $this->connexion->executeQuery('CALL MettreAJourCodePromo(?,?,?,?)',
[$id,$pourcentage,$nbr_use,$nom_code]    }
    }

    public function DeleteCode($id){
        $this->connexion->executeQuery('CALL SupprimerCodePromo(?)',
        [$id]);
    }
}
```

Imagions maintenant que nous voulons ajouter un code_promo, voici comment cela se passe dans le Contrôleur du code Promo:

```
 public function deleteCode(int $id, EntityManagerInterface  $entityManager,
ProduitRepository $repository, UserRepository $repository_user,CodePromoRepository
$repository_code_promo){
          //La méthode DeleteCode() appelle une procedure
          //Permettant de supprimer un code promo.
         $repository_code_promo->DeleteCode($id);
        return $this->redirectToRoute('app_admin_controlleur',
[$repository,$repository_user,$repository_code_promo]);
    }
```

C / Quatrième Design Pattern : Facade

1/ réponse à un problème.

Notre site web contient de multiples contrôleurs qui correspondent à de multiples fonctionnalités. Nous avons + de 10 Contrôleurs qui contiennent eux même plusieurs méthodes. Pour nous éviter de nous embrouiller, le mieux serait d'une faire une classe (une facade) qui contient tous les contrôleurs. Autrement dit, si nous voulons ajouter un produit en tant que Administrateur, J'appelle dans ma facace le controlleur AdminController et j'appelle sa méthode addProduct.

2/ Mise en œuvre:

Comme dit auparavant, La création de cette facade permet de rassembler tous les contrôleurs. Une facade contiendra donc tous les contrôleurs en attributs, et chaque méthode de notre classe Facace sera en réalité un appel de fonction d'un controlleur. Pour des raisons qui seront expliqué après, nous avons décidé de ne pas l'intégrer dans notre Diagramme UML, déjà très conséquent. En effet le rajouter ne représenterai réellement notre site.

3/ Développement en PHP:

Finalement, Le développement en PHP s'est révélée beaucoup plus complexe que prévu. En Effet Symfony n'est pas du tout fait pour pouvoir utiliser des designs paterns comme celui là puisque les redirections de routes sont faites directement dans les controlleurs. En d'autres termes, si je voulais par exemple afficher la page AboutUs. la fonction se trouve dans mon Contrôleur AboutUs et permet de afficher la vue. En rajoutant une facade, il est donc nécessaire de refaire une autre route qui va appeler l'autre route et donc à besoin d'un attribut de redirection de route....

Cela étant beaucoup plus contraignant à implémenter, nous avons donc décider de ne pas l'implémenter pour éviter le risque de nous perdre. **Vous retrouverez néanmoins dans ce rendu (sous le nom de** GlobalSiteManagerController.php) **, notre classe entièrement développée pour ce besoin.**