Mission 1 :

Objectifs et Contexte de l'Activité

L'objectif principal de cette itération était l'apprentissage et la mise en œuvre de **Doctrine**, l'outil de mapping objet-relationnel (ORM) intégré par défaut dans Symfony. Cette étape est cruciale car elle permet d'établir un pont entre le code applicatif en PHP et la base de données relationnelle. Au lieu de manipuler directement des requêtes SQL complexes, le développeur interagit avec des **entités** (objets PHP), ce qui simplifie la gestion de la persistance des données et assure une meilleure cohérence entre le schéma de la base de données et la logique métier.

Modélisation et Persistance des Données

Le travail a débuté par la création d'une classe métier, l'entité `Produit`, destinée à représenter les articles de l'application (libellé, prix, date de création).

- **Mapping et Métadonnées** : Grâce aux attributs PHP 8, chaque propriété de l'objet a été associée à une colonne spécifique en base de données, définissant ainsi les métadonnées de l'entité.
- **Synchronisation du Schéma** : L'utilisation du bundle `DoctrineMigrations` a permis de transformer automatiquement ces définitions d'objets en tables physiques dans MySQL. Ce processus garantit que la structure de la base de données évolue de manière sécurisée et synchronisée avec le code.
- **Logique de Construction** : La classe a été adaptée pour automatiser certains comportements, comme l'initialisation systématique de la date de création lors de l'instanciation d'un nouvel objet.

Manipulation des Objets via le Contrôleur

Une part importante de l'activité a consisté à implémenter les opérations fondamentales de gestion de données (CRUD) au sein du `ProduitController`.

- **Enregistrement et Modification** : L'utilisation de l'`EntityManagerInterface` permet de "persister" des objets, c'est-à-dire d'indiquer à Doctrine qu'un objet doit être sauvegardé ou mis à jour en base de données après l'exécution de la méthode `flush()`.
- **Récupération de Données** : La lecture des informations s'appuie sur le **Repository** associé à l'entité. Nous avons exploré plusieurs méthodes de recherche, allant de la simple récupération par identifiant (`find`) à l'utilisation du **DQL** (Doctrine Query Language) pour des requêtes personnalisées, comme le filtrage de produits par prix.
- **Simplification du Code** : L'introduction de l'`EntityValueResolver` (Doctrine Bridge) a démontré comment Symfony peut injecter automatiquement un objet

complet dans une méthode de contrôleur à partir d'un paramètre d'URL, réduisant ainsi considérablement le code répétitif.

Gestion des Relations entre Entités

L'activité s'est conclue sur la mise en œuvre d'une association de type **ManyToOne / OneToMany** entre les produits et leurs catégories.

Cette étape modélise la réalité métier où un produit appartient à une seule catégorie, tandis qu'une catégorie peut regrouper plusieurs produits. La commande `make:entity` a été réutilisée pour mettre à jour les classes existantes, ajoutant les attributs nécessaires et les méthodes de gestion de collection (`addProduit`, `removeProduit`). Malgré des contraintes d'intégrité référentielle rencontrées lors de la migration des données existantes, cette structure permet désormais de sauvegarder simultanément un produit et sa catégorie liée dans une transaction unique.

Cette mission pose les bases nécessaires pour aborder ultérieurement des relations plus complexes (ManyToMany) et l'utilisation de formulaires automatisés pour la saisie des données.

Mission 2 :

Modélisation de Relations Bidirectionnelles (ManyToMany)

L'évolution de l'application a nécessité la mise en place d'une gestion de recettes de cuisine. Ce cas d'usage illustre une relation **ManyToMany** : une recette peut contenir plusieurs produits, et un produit peut figurer dans plusieurs recettes.

- **Concept de Propriété** : Dans une relation bidirectionnelle, une entité est désignée comme "propriétaire" (ici `Recette`). C'est elle qui porte l'attribut `inversedBy`, tandis que l'entité "inverse" (`Produit`) utilise `mappedBy`. Cette distinction est essentielle pour que Doctrine sache quelle entité surveiller pour déclencher les mises à jour en base de données.
- **Abstraction de la Table de Liaison** : Contrairement au SQL traditionnel, il n'est pas nécessaire de créer une entité spécifique pour la table de jointure. Doctrine génère et gère la table intermédiaire `recette_produit` de manière transparente à travers les collections d'objets définies dans les classes PHP.

Maintenance et Intégrité du Schéma

Une phase de correction a été nécessaire pour stabiliser la base de données suite aux travaux précédents.

- **Résolution des Conflits de Migration** : Afin de respecter les contraintes d'intégrité (clés étrangères), une mise à jour manuelle des données existantes a été effectuée pour lier tous les produits orphelins à une catégorie valide avant d'appliquer les nouvelles structures.
- **Évolution du Schéma** : L'exécution des nouvelles migrations a permis de créer physiquement les tables `recette` et `recette_produit` tout en conservant la cohérence globale du système.

Industrialisation des Tests avec les Fixtures

L'introduction du `DoctrineFixturesBundle` marque une étape vers des méthodes de développement professionnelles. L'objectif est de permettre au développeur d'initialiser la base de données dans un **état connu et stable** avant chaque phase de test.

- **Génération de Données Massives** : Plutôt que de saisir les produits manuellement via des contrôleurs, nous avons automatisé la création d'un catalogue complet (Fruits, Légumes, Aromatiques, etc.) via la classe `ProduitFixtures`.
- **Logique Aléatoire** : Pour simuler une utilisation réelle, les fixtures incluent des algorithmes associant aléatoirement des produits aux recettes créées, permettant ainsi de tester la robustesse des jointures ManyToMany.
- **Gestion des Utilisateurs et Sécurité** : Les `UserFixtures` ont été configurées pour générer des comptes de test réalistes. L'utilisation de `Faker` pour les noms/téléphones et du `UserPasswordHasherInterface` garantit que les mots de passe sont correctement hachés, respectant les standards de sécurité de Symfony.
- **Purge et Persistance** : La commande `doctrine:fixtures:load` permet de réinitialiser totalement la base (purge) ou d'ajouter des données (`--append`), offrant une flexibilité totale pour le débogage.

Mission 3 :

Objectifs et Modularité des Formulaires

L'objectif central était de passer d'une manipulation de données via des méthodes de contrôleur vers une interface utilisateur interactive et sécurisée.

- **Création de Formulaires Types** : Plutôt que de construire les formulaires directement dans les contrôleurs, nous avons utilisé la commande `make:form` pour générer une classe `CategorieType`. Cette approche modulaire permet de dissocier la définition des champs de la logique de traitement, facilitant ainsi la réutilisation du formulaire dans différentes parties de l'application.
- **Mapping Sémantique** : Symfony fait le lien entre les propriétés de l'entité PHP (ex: un `string`) et des types sémantiques de formulaires (ex: `TextType`), qui sont

ensuite automatiquement rendus en HTML sous forme de balises appropriées (ex: `input type="text"`).

Affichage et Expérience Utilisateur

Le rendu visuel a été intégré dans les templates Twig en s'appuyant sur des fonctions dédiées qui simplifient la génération du code HTML.

- **Intégration Twig** : L'utilisation de fonctions telles que `form_start()`, `form_widget()` et `form_end()` permet de générer dynamiquement les balises de formulaire, les champs de saisie et les jetons de sécurité.
- **Filtres et Mise en Page** : La liste des catégories est affichée dans un tableau structuré avec Bootstrap, utilisant des filtres Twig comme `| length` pour compter les éléments et des messages "flash" pour informer l'utilisateur du succès de ses actions (ajout, modification ou suppression).

Sécurité et Validation des Données

La mission a mis un accent particulier sur la robustesse et la sécurité de l'application face aux saisies incorrectes ou malveillantes.

- **Protection CSRF** : Symfony intègre par défaut une protection contre les attaques de type *Cross-Site Request Forgery* en insérant un jeton aléatoire caché dans chaque formulaire. Pour les formulaires créés manuellement (comme celui de la suppression), nous avons dû générer et valider ce token explicitement dans le contrôleur.
- **Double Validation** :
    - **Côté Client** : Utilisation des attributs HTML5 comme `required` pour un premier contrôle immédiat dans le navigateur.
    - **Côté Serveur** : Utilisation d'assertions PHP 8 (attributs `#[Assert]`) au sein de l'entité `Categorie` (ex: `NotBlank`, `Length`). Cette validation serveur est indispensable car elle ne peut pas être contournée, contrairement à la validation client.

Gestion des Opérations CRUD

Le `CategorieController` a été enrichi pour traiter l'intégralité du cycle de vie d'une catégorie.

- **Ajout et Hydratation** : La méthode `handleRequest()` permet à Symfony de récupérer les données de la requête et d' "hydrater" automatiquement l'objet `Categorie`, qui est ensuite persisté en base de données via l'`EntityManager`.
- **Modification Contextuelle** : Pour l'édition, la vue Twig a été adaptée afin de transformer la ligne du tableau concernée en champ de saisie, permettant une modification directe sans changer de page.

- **Suppression Sécurisée** : Avant toute suppression, le contrôleur vérifie non seulement le jeton CSRF, mais aussi si la catégorie est vide. Si des produits sont encore liés à une catégorie, la suppression est bloquée afin de garantir l'intégrité référentielle des données.

Cette mission achève la mise en place d'une interface d'administration fonctionnelle, sécurisée et modulaire pour les catégories, posant les jalons pour la gestion plus complexe des produits et des recettes.