SYNTHÈSE SPRINT 1

La ferme **Maya** propose des fruits et légumes de saison issus d'une agriculture raisonnée ou biologique. Elle possède un site vitrine mais celui-ci est ancien et ne donne plus satisfaction.

La ferme a décidé de confier la refonte de son site internet à l'ESN Logma. Le développement de l'application se fera dans un cadre de travail agile SCRUM.

Dans ce premier sprint, il conviendra de mettre en place un système d'authentification afin que limiter l'accès à certaines données pour les utilisateurs.
Logma a choisi d'utiliser le framework **Symfony** pour l'ensemble des sprints.

Commandes Symfony :

Création du projet : **symfony new Maya --version=6.3 –webapp**

Création d'un contrôleur : php bin/console make:controller AccueilController

Lancer le serveur : symfony serve

Création de la base de donneés : **php bin/console doctrine:database:create**

Création d'une entité : php bin/console make:entity

Création du fichier de migration : php bin/console make:migration

Exécuter la migration : php bin/console doctrine:migrations:migrate

Installation bundle DoctrineFixturesBundle : composer require --dev orm-fixtures

Création de fixtures : php bin/console make:fixtures

Création de données aléatoire : composer require --dev fakerphp/faker

Charger et ajouter les données à la base de données : php bin/console doctrine:fixtures:load --group=UserFixtures –append

Création d'un formulaire d'authentification : php bin/console make:auth

Générer une interface CRUD : php bin/console make:crud User

Dans ce début de projet, la barre de menu puis les contrôleurs des pages Accueil, Produit et Catégorie furent créés. Le menu a été créé dans le fichier base.html.twig. (voir commande symfony)
Il faut relier le projet à sa base de données qui s'effectue dans le fichier .env, puis créer la base de données en ligne de commande. (voir commande symfony)

Il faut ensuite créer la classe user (en ligne de commande) en reprenant ces paramètres :

```
C:\wamp64\www\Maya>php bin/console make:user

 The name of the security user class (e.g. User) [User]:
 > User

 Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
 > yes

 Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
 > email

 Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be
checked/hashe
d by some other system (e.g. a single sign-on server).

 Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes

 created: src/Entity/User.php
 created: src/Repository/UserRepository.php
 updated: src/Entity/User.php
 updated: config/packages/security.yaml


  Success!


 Next Steps:
   - Review your new App\Entity\User class.
   - Use make:entity to add more fields to your User entity and then run make:migration.
   - Create a way to authenticate! See https://symfony.com/doc/current/security.html

C:\wamp64\www\Maya>
```

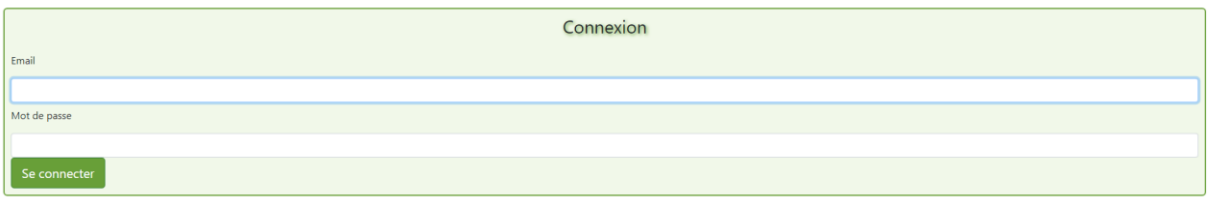
Grâce à cette commande, des fichiers sont créés et des modifications sont effectués dans d'autres fichiers.

Puis il faut créer une entité grâce à sa commande dédiée. Une fois l'entité crée, il faut générer un fichier de migration et l'exécuter. (voir commande symfony)

Par la suite, il faut créer les fixtures pour les utilisateurs. Mais avant cela, il faut installer le bundle DoctrineFixturesBundle. (voir commande symfony)
La commande composer require --dev fakerphp/faker permet de générer des données facilement, en l'occurrence ici ce sera pour les utilisateurs. Il faut ensuite lancer les fixtures et les ajouter à la base de données. (voir commande symfony).

Il faudra aussi créer un formulaire d'authentification. Puis gérer les autorisations, attribuer les rôles. Il faudra attribuer la gestions des utilisateurs aux admins, nous utiliserons la commande make:crud.

Formulaire de connexion :



SYNTHÈSE SPRINT 2

La ferme **Maya** propose des fruits et légumes de saison issus d'une agriculture raisonnée ou biologique. Elle possède un site vitrine mais celui-ci est ancien et ne donne plus satisfaction.

La ferme a décidé de confier la refonte de son site internet à l'ESN Logma. Le développement de l'application se fera dans un cadre de travail agile SCRUM.

Dans ce second sprint, il conviendra d'utiliser un des composants de Symfony qui est l'ORM Doctrine.

Commandes Symfony :

Création d'une entité : **php bin/console make:entity**

Création du fichier de migration : php bin/console make:migration

Migrer le fichier vers la base de données : php bin/console doctrine:migrations:migrate

Lancer et ajouter les données à la base de données : **php bin/console doctrine:fixtures:load --group=ProduitFixtures –append**

Création d'un formulaire : php bin/console make:form

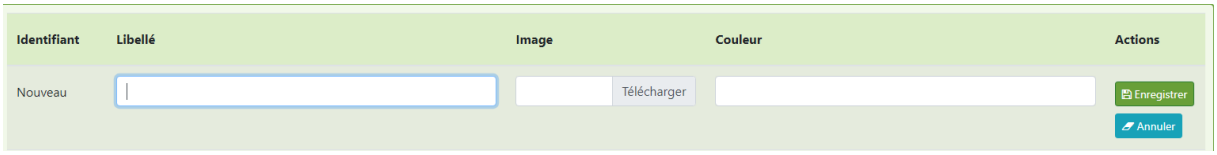
Pour ce début de sprint, nous allons créer les produits. Il faudra créer son entité, son contrôleur.
Puis il faut faire une association entre le produit et la catégorie. Cela s'effectue lors de la modification d'une entité (même commande que pour la création, puis entrer le nom d'une entité déjà inscrite), puis créer un attribut et choisir « relation » pour le type. Il existe 4 types de relation, en voici l'exemple entre produit et catégorie :

```
What type of relationship is this?
------------ ---------------------------------------------------------------
 Type         Description
------------ ---------------------------------------------------------------
 ManyToOne    Each Produit relates to (has) one Categorie.
              Each Categorie can relate to (can have) many Produit objects

 OneToMany    Each Produit can relate to (can have) many Categorie objects.
              Each Categorie relates to (has) one Produit

 ManyToMany   Each Produit can relate to (can have) many Categorie objects.
              Each Categorie can also relate to (can also have) many Produit objects

 OneToOne     Each Produit relates to (has) exactly one Categorie.
              Each Categorie also relates to (has) exactly one Produit.
------------ ---------------------------------------------------------------
```

Par la suite, il faudra créer les recettes et faire une association avec les produits.
Afin d'avoir des produits, nous créerons une classe fixtures et nous y ajouterons manuellement les produits dedans. Il ne faudra pas oublier de lancer les fixtures créées. (voir commande Symfony)

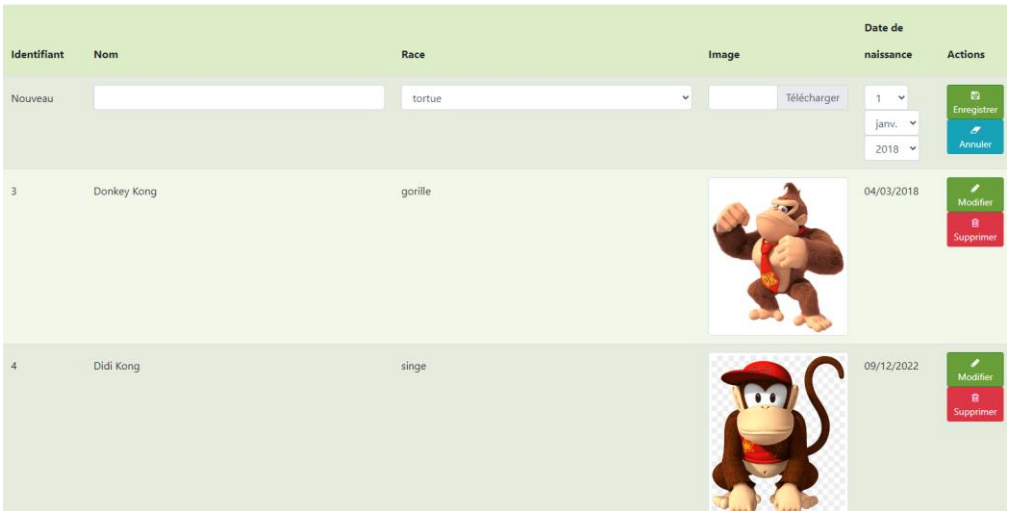
Il faut aussi créer la vue pour les catégories, qui s'effectue dans le fichier index.html.twig du dossier categorie se trouvant dans les templates.

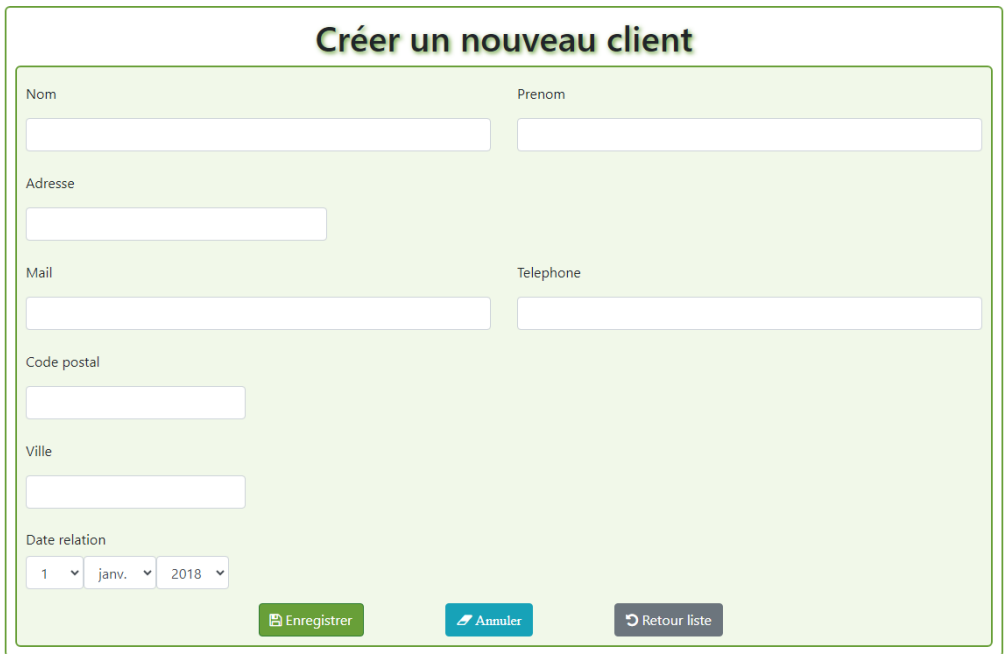
Il sera nécessaire d'avoir un formulaire pour la page catégorie afin de pouvoir ajouter de nouvelle catégorie. (voir commande Symfony)
Affichage du formulaire :



Il faudra ajuster le code de la vue afin que la page affiche ce formulaire.

A la suite de la page catégorie, il faut créer les pages évènements, clients, fournisseurs, races d'animaux et animaux, tout en reprenant le même schéma que pour la création de la page catégorie. Toutes ces pages comprendront des données ainsi qu'un formulaire d'ajout.
Aperçu de certaines pages :





SYNTHÈSE SPRINT 3

La ferme **Maya** propose des fruits et légumes de saison issus d'une agriculture raisonnée ou biologique. Elle possède un site vitrine mais celui-ci est ancien et ne donne plus satisfaction.

La ferme a décidé de confier la refonte de son site internet à l'ESN Logma. Le développement de l'application se fera dans un cadre de travail agile SCRUM.

Dans ce troisième sprint, il conviendra tout d'abord de mettre en place la gestion des produits et des recettes, suivi de la mise en place des recherches multi-critères à la gestion des clients et des fournisseurs. Ajouter la pagination à tous les objets de gestion, et afficher la liste des catégories sous forme de card sur la page d'accueil.

Commandes Symfony :

Installer le bundle pour la pagination avec composer : **composer require knplabs/knp-paginator-bundle**

Création d'un formulaire : **php bin/console make:form**

Tout d'abord, il faut créer le formulaire pour l'ajout, la modification et la suppression de la donnée. Nous commençons avec la page des produits. Toutes les méthodes concernant ces actions sont ajoutées dans le contrôleur.
Il faudra ajouter les mêmes méthodes à la page recettes, en adaptant selon les besoins.

Puis il faut créer la recherche pour ces produits, pour ainsi pouvoir trouver plus facilement et plus rapidement ce que l'on cherche. Il faut créer un fichier ProduitRechercheType.php. Des méthodes sont à ajoutées dans le ProduitRepository afin que les recherches s'effectuent correctement.

Enfin, pour avoir un meilleur visuel de la page, il est préférable d'y ajouter une pagination. Un ajout qui se fait facilement via le contrôleur comme ci-dessous :

```
    $lesProduits = $paginator->paginate(
        $repository->findAllByCriteria($produitRecherche),
        $request->query->getint('page', 1),
        5
    );
} else {
    // lire les produits
    if ($session->has("ProduitCriteres")) {
        $produitRecherche = $session->get("ProduitCriteres");
        $lesProduits = $paginator->paginate(
            $repository->findAllByCriteria($produitRecherche),
            $request->query->getint('page', 1),
            5
        );
        $formRecherche = $this->createForm(ProduitRechercheType::class, $produitRecherche);
        $formRecherche->setData($produitRecherche);
    } else {
        $p = new ProduitRecherche();
        $lesProduits = $paginator->paginate(
            $repository->findAllOrderByLibelle($p),
            $request->query->getint('page', 1),
            5
        );
```

Ainsi que quelques modifications dans le Repository, et dans l'index.
(Repository : remplacer « return $qb->getQuery() ; » par « return $query »)
(Index : ajouter une balise contenant « {{ knp_pagination_render(lesProduits) }} »)

Ensuite, il faut reproduire tous ces ajouts pour la page clients, et fournisseurs, c'est-à-dire pouvoir gérer, faire la recherche multi-critère et la pagination sur ces 2 pages. Il faut reprendre les mêmes instructions en adaptant selon les besoins.

La pagination sera faite sur les autres pages aux objets de gestion.

Il ne reste plus que l'affichage de la liste des catégories sous forme de card sur la page d'accueil à mettre en place. Pour se faire, il faut rajouter une partie de code dans la vue de la page comme ceci :

```
<h1>Les Catégories</h1>

<div class="row row-cols-1 row-cols-md-3 g-4" style="margin-top: 20px">
    {% for categorie in lesCategories %}
        <div class="col card" style="margin-bottom: 25px">

            <a href="{{ path('app_produit')}}" class="card"><img class="img-thumbail img-fluid max" src="{{ vich_uploader_asset(categorie, 'imageFichier') }} " />

                <div class="card-body">
                    <h5 class="card-title text-center">{{ categorie.libelle }}</h5>
                </div>
            </a>
        </div>
    {% endfor %}
</div>
```

Il est possible de les personnaliser via un fichier CSS.

SYNTHÈSE SPRINT 4

La ferme **Maya** propose des fruits et légumes de saison issus d'une agriculture raisonnée ou biologique. Elle possède un site vitrine mais celui-ci est ancien et ne donne plus satisfaction.

La ferme a décidé de confier la refonte de son site internet à l'ESN Logma. Le développement de l'application se fera dans un cadre de travail agile SCRUM.

Dans ce quatrième sprint, il conviendra d'ajouter certaines fonctionnalités au site afin de le rendre plus attractif. Il faudra ajouter une case indiquant la météo sur la page d'accueil via une API REST, puis d'afficher à la demande, la liste des recettes pour chaque produit (en utilisant Ajax).

Commandes Symfony :

Tout d'abord, pour pouvoir mettre en place une API REST affichant la météo, il faut avoir une API. Nous utiliserons OpenWeatherMap qui fournit régulièrement un énorme volume d données météorologiques.
En premier temps, il faudra se créer un compte sur leur site pour récupérer une clé d'API que l'on intégrera dans notre code. Puis il suffira de rajouter quelques lignes de codes dans le contrôleur de l'accueil et dans la page index comme ci-dessous :

Dans le contrôleur :

```php
// récupérer la variable d'environnement désignant l'URl de l'API
$urlAPI = $_SERVER['URL_API_OWM'];

// Initialiser une session CURL
$clientURL = curl_init();
// Récupérer le contenu de la page
curl_setopt($clientURL, CURLOPT_RETURNTRANSFER, 1);
// Transmettre l'URL
curl_setopt($clientURL, CURLOPT_URL, $urlAPI);
// Exécutez la requête HTTP
$reponse = curl_exec($clientURL);
// Fermer la session
curl_close($clientURL);
// Récupérer les données au format JSON
$donneesTemps = json_decode($reponse);

$dateJour = new \DateTime('now', new \DateTimeZone('Europe/Paris'));



return $this->render('accueil/index.html.twig', [
    'lesCategories' => $lesCategories,
    'donneesTemps' => $donneesTemps,
    'dateJour' => $dateJour,
]);
```

Dans l'index :

```
<div class="conteneur-meteo">
    <h2>Le temps à {{ donneesTemps.name }}</h2>
    <div>Le {{ dateJour | date("d/m/Y à H\hi") }}</div>
    <div class="temps-previsions">{{donneesTemps.weather[0].description | capitalize }}</div>
    <div class="temps-previsions">
        <img src="http://openweathermap.org/img/w/{{ donneesTemps.weather[0].icon }}.png" class="temps-icone"/>
        {{ donneesTemps.main.temp_max | number_format(2, ',') | trim("0", "right") | trim(',') }}&deg;C
        <span class="temperature">{{ donneesTemps.main.temp_max | number_format(2, ',') | trim("0", "right") | trim(',') }}&deg;C</span>
    </div>
    <div>Humidité : {{ donneesTemps.main.humidity }}%</div>
    <div>Vent : {{ donneesTemps.wind.speed }}km/h</div>
</div>
```

Une fois la météo affichée, il faut passer à l'affichage des recettes sur chaque produit, en utilisant Ajax. Il y a tout d'abord quelques fichiers à insérer dans le projet, concernant jquery. Une fois mis en place, le code principal se trouve dans l'index de la page concernée, à commencer par cette ligne de code qui fera appel à la méthode afficherMesRecettes :

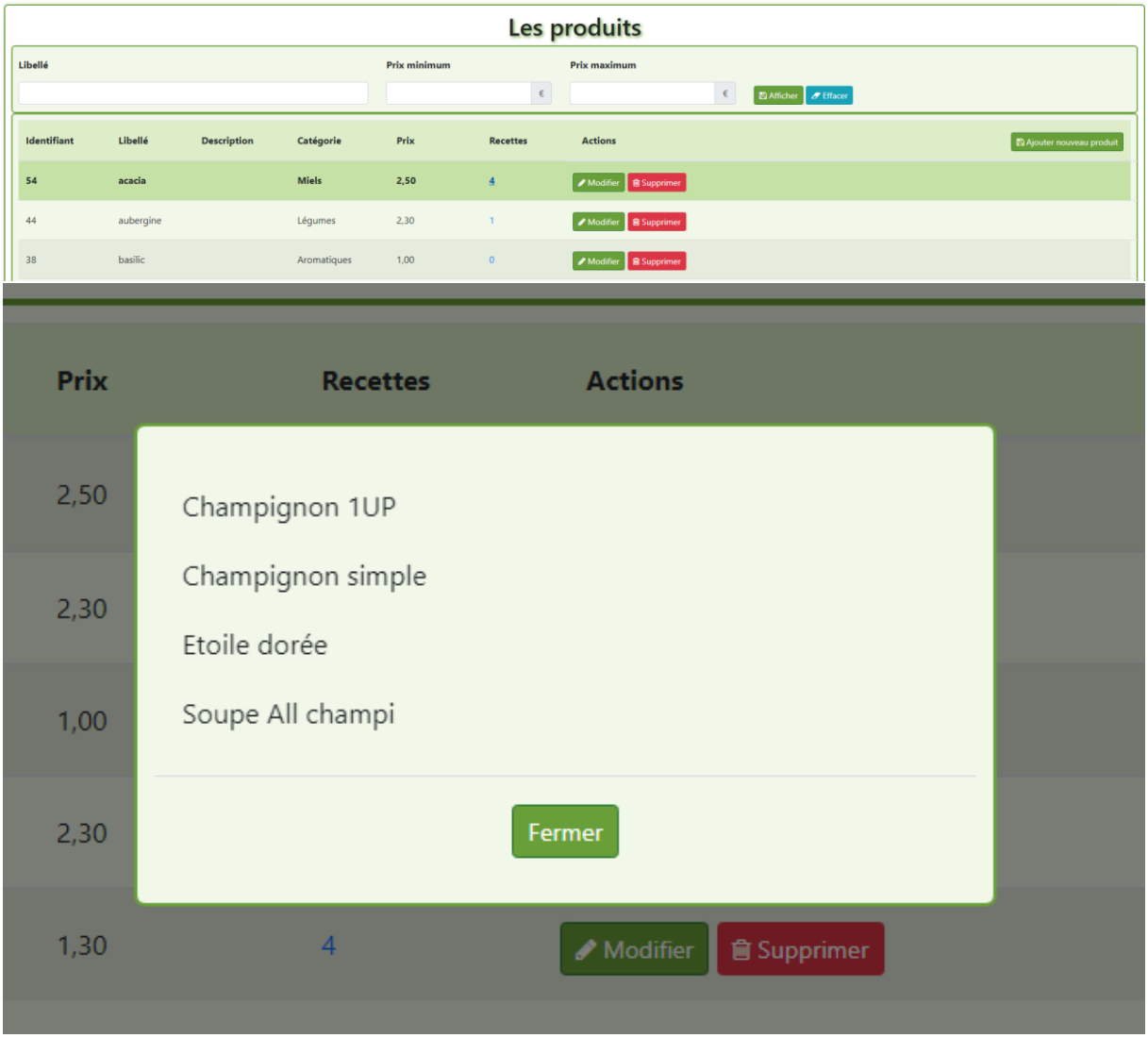
```
{% for key, produit in lesProduits %}
    <tr>
        <td>{{ produit.id }}</td>
        <td>{{ produit.libelle }}</td>
        <td>{{ produit.description }}</td>
        <td>{{ produit.categorie.getLibelle() }}</td>
        <td>{{ produit.getPrix() | number_format(2, ',', ' ') }}</td>
        <td>
            <a data-toggle="modal" href="#" onclick="afficherMesRecettes({{ produit.id }});" >{{ produit.getRecettes() | length }}</a>
        </td>
        <td
```

Une méthode que l'on définira plus bas dans un block javascripts :

```
{% block javascripts %}
    <script src="{{ asset('assets/jquery/jquery-3.6.0.min.js') }}"></script>
    <script src="{{ asset('assets/lib/bootstrap-4.4.1-dist/js/bootstrap.min.js') }}"></script>
    <script>
        function afficherMesRecettes(idProduit) {
            $(document).ready(function() {
                $.ajax({
                    url: '{{ path('ajax_recettes_produit') }}',
                    type: "POST",
                    dataType: "json",
                    data: {
                        "idProduit": idProduit
                    },
                    async: true,
                    success: function (lesRecettes) {
                        var chaineHtml = "";
                        for (var i = 0; i < lesRecettes.length; i++) {
                            chaineHtml = chaineHtml + "<div>" + lesRecettes[i].nom + "</div>";
                        }
                        $('#mesRecettesContenu').html(chaineHtml);
                        $('#mesRecettesModal').modal('show');
                    },
                    error: function(jqXHR){         // Fonction à appeler si la requête échoue
                        $("#mesRecettesContenu").html('<div class="text-danger">Erreur n°128</div>');
                    }
                })
            });
        }
    </script>

{% endblock %}
```

Il faudra ajouter une partie de code dans le contrôleur produit afin d'ajouter la méthode ajaxRecettesProduit, puis une requête DQL dans le repository de recette afin de récupérer ce que l'on souhaite.

Voici un apperçu :



SYNTHÈSE SPRINT 5

La ferme **Maya** propose des fruits et légumes de saison issus d'une agriculture raisonnée ou biologique. Elle possède un site vitrine mais celui-ci est ancien et ne donne plus satisfaction.

La ferme a décidé de confier la refonte de son site internet à l'ESN Logma. Le développement de l'application se fera dans un cadre de travail agile SCRUM.

Dans ce cinquième et dernier sprint, il faudra ajouter des photos sur les pages catégories, produits, animaux et mettre en place une gestion des données via des opérations CRUD.
Puis sur la page d'accueil, afficher les catégories avec leurs images correspondantes en ajoutant les prochains et précédents évènements afin de les mettre en valeur.
Ensuite, il faut faire une page de statistique des catégories avec plusieurs informations les concernant.
Il faudra créer une page de produit en détail afin d'afficher les recettes d'un produit choisi au préalable via une liste déroulante.
Enfin, ajouter la trace des connexions réussis et en échec en utilisant le logger Symfony.

Commandes Symfony :

Installer le bundle VichUpload : **symfony composer require vich/uploader-bundle**

Mettre à jour le schéma de la base de données : **symfony console doctrine:schema:update --force**

Dans un premier temps, il faut ajouter les photos à certaines pages afin de les embellir. Il faut installer un bundle pour pouvoir télécharger des photos sur le site pour pouvoir les mettre. (voir commande Symfony)
Il faut adapter certains fichiers afin que le bundle fonctionne. Puis ajouter des lignes de code dans les pages où nous faisons appelle à cette fonctionnalité afin qu'elle fonctionne. Il faudra mettre à jour le schéma de la base de données car nous faisons des modifications dans certaines classes.
Il faut ensuite ajouter un champ dans le formulaire afin que nous puissions ajouter une photo au moment où nous utilisons le formulaire. Ne pas oublier de l'ajouter dans l'index afin d'afficher la photo. Il faut répéter ces opérations pour les pages suivantes : catégories / produits / animaux.
Il faut aussi ajouter les opérations CRUD sur ces pages.

Dans un second temps, il faut afficher sur la page d'accueil, les catégories et les images que nous leurs avons attribué dans la page catégorie. Tout se fait dans la page index, mais nous pouvons personnaliser l'affichage via un fichier CSS.

```
<div class="row row-cols-1 row-cols-md-3 g-4" style="margin-top: 20px">
    {% for categorie in lesCategories %}
        <div class="col card" style="margin-bottom: 25px">

            <a href="{{ path('app_produit')}}" class="card"><img class="img-thumbail img-fluid max" src="{{ vich_uploader_asset(categorie, 'imageFichier') }} " />

                <div class="card-body">
                    <h5 class="card-title text-center">{{ categorie.libelle }}</h5>
                </div>
            </a>
        </div>
    {% endfor %}
</div>
```

En troisième temps, il faut faire la page de statistique qui regroupe plusieurs moyennes sur les catégories. Cette page se crée simplement comme les autres, en utilisant la commande du contrôleur. Puis il faut faire une requête DQL afin d'avoir les moyennes que l'on veut, et on affiche sur l'index.

Par la suite, il faut faire la page de produit détail qui permet d'afficher les recettes qui contiennent le produit choisi en paramètre. Il faut créer la page avec la commande contrôleur. Le code principal s'écrit dans l'index. Il faut faire la liste déroulante pour afficher tous les produits, puis une requête ajax pour afficher les recettes.

Enfin, il reste à ajouter un journal pour tracer les connexions réussis et en échec. Il y a quelques lignes de code à ajouter.

Dans le contrôleur sécurité :

```php
if ($error) {
    $logger->error('Un utilisateur a essayé de se connecter mais a échoué.');
} else {
    $logger->info('Un utilisateur a réussi à se connecter.');
}
```

Dans le monolog.yaml :

```yaml
    type: stream
    path: "%kernel.logs_dir%/%kernel.environment%.log"
    level: debug
    channels: ["!event"]
```

Et les connexions réussis se trouvent dans le chemin suivant : /var/cache/log

Le site est au complet, toutes les fonctionnalités ont été ajoutées.