# Table des matières



## I- Problématique

Pour le sprint 5, M. Vigne, en tant que Product Owner, a défini les fonctionnalités essentielles pour finaliser la version 1.0 de l'application Maya. Ces user stories simplifiées orientent le développement vers des aspects clés de l'expérience utilisateur et de la gestion des données.

Les objectifs comprennent l'ajout de photos aux catégories, aux produits et aux animaux, avec la mise en place des opérations CRUD. Sur la page d'accueil, l'affichage des trois derniers événements passés et des trois prochains événements permettra de les mettre en valeur.

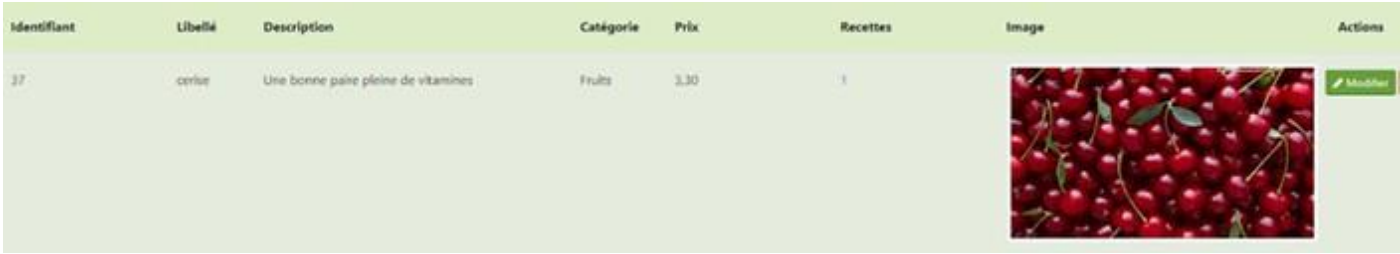
Une nouvelle page statistiques sera intégrée, affichant par catégorie le nombre de produits, le prix minimum, le prix maximum et le prix moyen des produits. Une fonctionnalité additionnelle consistera à créer une page permettant de sélectionner un produit dans une liste déroulante et d'afficher le détail de toutes les recettes utilisant ce produit, présentées par durée croissante. La liste des recettes sera obtenue via une requête Ajax.

Enfin, une trace des connexions réussies et en échec sera ajoutée en utilisant le logger Symfony, renforçant ainsi la sécurité et la traçabilité au sein de l'application. Ces développements stratégiques reflètent l'engagement envers l'achèvement de la version 1.0 tout en enrichissant les fonctionnalités offertes par l'application Maya.

## II- Solutions proposées

### 1) Mission 1

- Ajouter une photo aux catégories, aux produits et aux animaux et gérer les opérations CRUD *Exemple des produits*




Dans un premier temps nous avons installé le bundle VichUploader

Après cela il a fallu adapter le fichier

‘config/packages/vich_uploader.yaml’ :

```yaml
vich_uploader:
    db_driver: orm

    mappings:
        categories:
            uri_prefix: /images/categories
            upload_destination: '%kernel.project_dir%/public/images/categori
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
        animaux:
            uri_prefix: /images/animaux
            upload_destination: '%kernel.project_dir%/public/images/animaux'
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
        produits:
            uri_prefix: /images/produits
            upload_destination: '%kernel.project_dir%/public/images/produits
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
```

Par la suite on a indiqué à la classe catégorie qu'elle est gérée par le module VichUploader :

```php
use Symfony\Component\HttpFoundation\File\File;
use Vich\UploaderBundle\Mapping\Annotation as Vich;

#[ORM\Entity(repositoryClass: ProduitRepository::class)]
#[Vich\Uploadable]
class Produit
{
```

```php
#[Vich\UploadableField(mapping: 'produits', fileNameProperty: 'imageNom', size: 'imageTaille')]
private ?File $imageFichier = null;

#[ORM\Column(nullable: true)]
private ?string $imageNom = null;

#[ORM\Column(nullable: true)]
private ?int $imageTaille = null;

#[ORM\Column(nullable: true)]
private ?\DateTimeImmutable $imageDateMaj = null;
```

```php
/**
 *
 * @param File|\Symfony\Component\HttpFoundation\File\UploadedFile|null $imageFile
 */
public function setImageFichier(?File $imageFichier = null): void
{
    $this->imageFichier = $imageFichier;

    if (null !== $imageFichier) {
        $this->imageDateMaj = new \DateTimeImmutable();
    }
}

public function getImageFichier(): ?File
{
    return $this->imageFichier;
}

public function setImageNom(?string $imageNom): void
{
    $this->imageNom = $imageNom;
}

public function getImageNom(): ?string
{
    return $this->imageNom;
}

public function setImageTaille(?int $imageTaille): void
{
    $this->imageTaille = $imageTaille;
}

public function getImageTaille(): ?int
{
    return $this->imageTaille;
}
```

Après cela nous mettons à jour la BDD

A la suite de cela nous avons ajouté un nouveau champ au formulaire dans ‘Maya/src/Form/CategorieType.php’ :

```php
->add('imageFichier', VichImageType::class, [
    'required' => false,
]);
```

Puis adaptons la vue

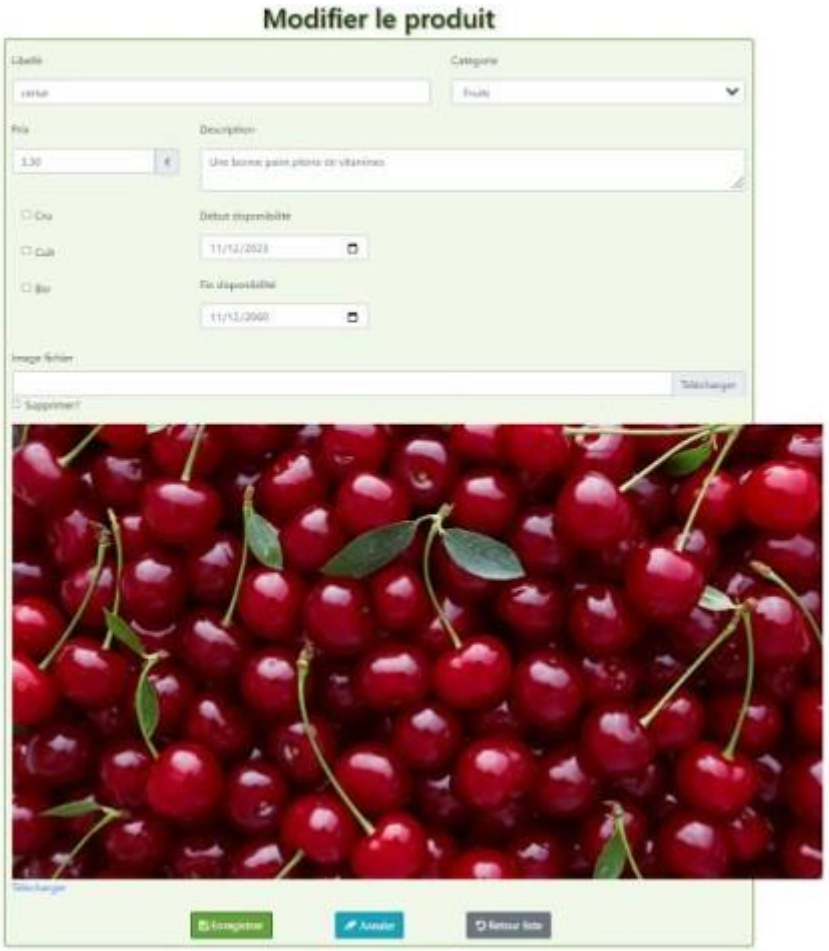
```
                        <td class="col-md-2">{{
form_widget(formCreation.imageFichier) }} {{
form_errors(formCreation.imageFichier) }} </td>
```

Nous modifions le CSS pour franciser le tout.

```css
.custom-file-input:lang(en) ~ .custom-file-label::after {
  content: "Browse";
}

.custom-file-input:lang(fr) ~ .custom-file-label::after {
  content: "Télécharger";
}
```

Le rendu :




## 2) Mission 2

- Dans le contrôleur de l’accueil

Nous récupérons les 3 derniers et les 3 prochains événements depuis la base de données.

```php
#[Route('/accueil', name: 'app_accueil')]
public function index(CategorieRepository $cRepository, EvenementRepository $eRepository): Response
{

    // 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'));

    $lesCategories = $cRepository->findAll();

    $lesEvenementsPasses= $eRepository->findByEvenementsPasses(3);
    $lesEvenementsFutures = $eRepository->findByEvenementsFutures(3);

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

Nous adaptons le repository

```php
public function findByEvenementsPasses($limit)
{
    return $this->createQueryBuilder('e')
        ->where('e.date < :now')
        ->setParameter('now', new \DateTime())
        ->orderBy('e.date', 'DESC')
        ->setMaxResults($limit)
        ->getQuery()
        ->getResult();
}
```

Puis la vue

```twig
<h1>Actualités</h1>
<h2>Précédents évènements</h2>
<div class="row row-cols-1 row-cols-md-3 g-4" style="margin-top: 20px">
    {% for key, evenement in lesEvenementsPasses %}
        <div class="card" style="margin-bottom: 25px;">
            <img class="card-img-top" src="{{ vich_uploader_asset(evenement, 'imageFichier') }}">
            <div class="card-body">
                <h5>{{ evenement.titre }}</h5>
                <p>{{ evenement.description }}
                <br>{{ evenement.date|date('d/m/Y à H:i') }}</p>
            </div>
        </div>
    {% endfor %}
</div>

<h2>Événements futurs</h2>
<div class="row row-cols-1 row-cols-md-3 g-4" style="margin-top: 20px">
    {% for evenement in lesEvenementsFutures %}
        <div class="card" style="margin-bottom: 25px;">
                <img class="card-img-top" src="{{ vich_uploader_asset(evenement, 'imageFichier') }}">
                <div class="card-body">
                    <h5>{{ evenement.titre }}</h5>
                    <p>{{ evenement.description }}
                    <br>{{ evenement.date|date('d/m/Y à H:i') }}</p>
                </div>
            </div>
    {% endfor %}
</div>
```

## 3) Mission 3

- Ajouter une page statistiques qui affiche par catégorie, le nombre de produits, le prix minimum, le prix maximum et le prix moyen des produits

Nous adaptons le contrôleur des statistiques

```php
class StatistiqueController extends AbstractController
{
    #[Route('/statistique', name: 'app_statistique_categorie')]
    public function index(Request $request, CategorieRepository $repository, PaginatorInterface $paginator): Response
    {
        $lesCategories= $paginator->paginate(
            $repository->findAllByStats(),
            $request->query->getInt('page',1),
            5
        );
        return $this->render('statistique/index.html.twig', [
            'lesCategories' => $lesCategories,
        ]);
    }
}
```

Puis nous utilisons le repository pour effectuer des requêtes personnalisées.

```php
public function findAllByStats(): Query
{
    $qb = $this->createQueryBuilder('c');
    $qb->select('c.libelle, count(p.id) as nbProduits, MIN(p.prix) as prixMin, MAX(p.prix) as prixMax')
    ->innerJoin("c.produits", "p")
    ->groupBy('c.libelle');

    return $qb->getQuery();
}
```

Enfin nous passons les statistiques à la vue

```twig
{% block title %}Statistiques{% endblock %}

{% block body %}
    <div class="col-md-12 mt-3 contenu-blanc">
        <h1>Statistiques : catégorie</h1>
        <div class="contenu">
            <table class="table table-striped table-advance table-hover">
                <thead>
                    <tr class="bg-entete">
                        <th class="col-md-3">Libellé</th>
                        <th class="col-md-3">Nombre de produits</th>
                        <th class="col-md-3">Prix minimum</th>
                        <th class="col-md-3">Prix maximum</th>
                    </tr>
                </thead>
                <tbody>
                    {% for categorie in lesCategories %}
                        <tr>
                            <td>{{categorie.libelle}}</td>
                            <td>{{categorie.nbProduits}}</td>
                            <td>{{categorie.prixMin}}</td>
                            <td>{{categorie.prixMax}}</td>
                        </tr>
                    {% endfor %}
                </tbody>
            </table>
        </div>
    </div>
{% endblock %}
```

## 4) Mission 4

- Ajouter une page qui permet de sélectionner un produit dans une liste déroulante et d'afficher le détail de toutes les recettes utilisant ce produit en les présentant par durée croissante. La liste des recettes sera obtenue via une requête Ajax.

Nous adaptons le contrôleur des recettes

```php
#[Route('/recette/ajaxrechercherecettesproduit', name: 'ajax_recherche_recettes_produit')]
public function ajaxRechercheRecettesProduit(Request $request, RecetteRepository $repository): Response
{
    $idProduit = $request->request->get('idProduit');
    $lesRecettes = $repository->findNameByProduitByTempsPrep($idProduit);
    $response = new Response(json_encode($lesRecettes));
    $response->headers->set('Content-Type', 'application/json');

    return $response;
}

#[Route('/recette/rechercheRecettes', name: 'app_recette_rechercheRecettes')]
public function rechercheRecettes(ProduitRepository $repository): Response
{
    $lesProduits=$repository->findProduitsWithRecette();

    return $this->render('recette/rechercheRecettes.html.twig', [
        'lesProduits' => $lesProduits,
    ]);
}
```

Nous faisons les requêtes dans les repository

```php
/**
 * @return Produit[]
 */
public function findProduitsWithRecette(): array
{
    $entityManager = $this->getEntityManager();

    $qb = $this->createQueryBuilder('p')
        ->orderBy('p.libelle', 'ASC')
        ->where('SIZE(p.recettes) != 0');

    return $qb->getQuery()->getResult();
}
```

```php
public function findNameByProduitByTempsPrep($idProduit): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery(
        'SELECT r.nom, r.tempsPreparation, r.description
        FROM App\Entity\Recette r
        JOIN r.produits p
        WHERE p.id = :idProduit
        ORDER BY r.tempsPreparation ASC'
    )->setParameter('idProduit', $idProduit);

    return $query->getResult();
}
```

Enfin nous modifions la vue et ajoutons le script JS
Comme cette fonctionnalité se trouve dans une page distincte, nous rajoutons une page dans le dossier ‘recette’ des vues

```twig
{% block body %}
<div class="row justify-content-center">
    <div class="col-md-5">
        <select id="lstProduits" class="form-select mb-3" onchange="afficherRechercheRecettes()">
            <option selected>Choix du produit</option>
            {% for produit in lesProduits %}
                <option value="{{produit.id}}">{{produit.libelle}}</option>
            {% endfor %}


        </select>
        <div class="container-fluid contenu">
            <table id="mesRecettesContenu" class="table table-striped table-advance table-hover">
                <thead>
                    <tr class="bg-entete">
                        <th class="col-md-2">Recette</th>
                        <th class="col-md-4">Description</th>
                        <th class="col-md-3">Temps de préparation</th>
                    </tr>
                </thead>
            </table>
        </div>
    </div>
<div>
{% endblock %}
```

```twig
{% 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 afficherRechercheRecettes() {
            $(function () {
                $.ajax({
                    url: '{{ path('ajax_recherche_recettes_produit') }}',
                    type: 'POST',
                    dataType: 'json',
                    data: {
                        'idProduit': $('#lstProduits').val()
                    },
                    success: function (lesRecettes) {
                        $('#mesRecettesContenu tbody').empty();

                        $.each(lesRecettes, function (i, recette) {
                            var chaineHtml =
                                `<tr>
                                    <td>${recette.nom}</td>
                                    <td>${recette.description}</td>
                                    <td>${recette.tempsPreparation} min</td>
                                </tr>`;
                            $('#mesRecettesContenu tbody').append(chaineHtml);
                        });
                    }
                });
            });
        }
    </script>


{% endblock %}
```

## 5) Mission 5

- Ajouter la trace des connexions réussies et en échec en utilisant le logger

Symfony. Il faut d'abord ajouter la librairie 'Monolog' au projet

Nous créons 'SecurityController.php'

```php
class SecurityController extends AbstractController
{
    #[Route(path: '/login', name: 'app_login')]
    public function login(AuthenticationUtils $authenticationUtils, LoggerInterface $connectionLogger): Response
    {
        // if ($this->getUser()) {
        //     return $this->redirectToRoute('target_path');
        // }
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();
        if ($error){
            $connectionLogger->error('User {lastUsername} : {error}', [
                'lastUsername' => $lastUsername,
                'error' => $error->getMessageKey(),
            ]);
        }
        return $this->render('security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error]);
    }

    #[Route(path: '/logout', name: 'app_logout')]
    public function logout(): void
    {
        throw new \LogicException('This method can be blank - it will be intercepted by the logout key on your firewall.');
    }
}
```

Un dossier 'Security' a été crée, nous ajoutons le fichier

‘LoginFormAuthentificator.php’

## III- Liste des commandes

| Commandes Description | |
|---|---|
| symfony composer require vich/uploader-bundle’ | Installe ‘VichUploader’ dans le projet, afin de pouvoir télécharger, stocker et afficher des images |
| symfony console doctrine:schema:update -- force | Force la mise à jour de la BDD |
| composer require symfony/monolog-bundle | Installe le librairie ‘Monolog’, qui est une des librairies PHP les plus populaires pour créer et gérer les logs (recommandé par Symfony) |

## IV- Conclusion

Ces morceaux de code forment un ensemble qui fait fonctionner les différentes parties mentionnées dans le compte rendu du sprint 5.

Ils comprennent des morceaux de code qui disent à Symfony quoi faire (contrôleurs Symfony), comment afficher les pages web (vues Twig), comment interroger la base de données (requêtes Doctrine), comment charger des informations en temps réel sans actualiser la page (requêtes Ajax avec JavaScript), et comment enregistrer des informations importantes comme les connexions (service de logging de Symfony).

Ces éléments travaillent ensemble pour créer des fonctionnalités spécifiques dans l'application.