

Dossier Projet

Titre RNCP – Développeur Web / Web Mobile

Leliard René

Studi – Promotion Novembre/Décembre 2024

Table des matières

2 => A propos de moi

3 => Compétences du référentiel couverte par le projet

4 => Présentation du projet

4 => Cahier des charges

7 => Contraintes

8 => Technologies, outils, Framework

10 => Gestion de projet

11 => Conception

16 => Charte graphique

17 => Wireframe et maquettes

18 => Installation et configuration de l’environnement de travail

22 => Développement

40 => Mailling

44 => Sécurité

47 => Déploiement

51 => Docker

59 => Annexes

A propos de moi

Je m’appelle Leliard René, j’ai 40 ans et je suis passionné par le développement web. Après plusieurs années d’expérience dans différents domaines professionnels, j’ai décidé de me reconvertir et de suivre une formation en développement web et web mobile afin de me spécialiser dans ce secteur en constante évolution. Cette décision est motivée par mon intérêt croissant pour les nouvelles technologies et le développement d’applications qui répondent aux besoins des utilisateurs.

Grâce à cette formation, j’ai acquis des compétences solides en front-end et back-end, en utilisant des technologies comme HTML, CSS, JavaScript, Symfony, PHP, ainsi que des bases de données relationnelles et non relationnelles. Mon objectif est de continuer à développer mes compétences tout en mettant à profit mon expérience professionnelle pour concevoir et déployer des solutions web robustes et sécurisées.

Le projet que je présente ici est le fruit de ma formation et de mes recherches. Il reflète mon engagement à produire un travail de qualité, tout en respectant les contraintes techniques et les attentes des utilisateurs finaux. Bien que ce projet soit une réalisation dont je suis fier, je reconnais qu’il reste encore des axes d’amélioration, notamment en matière d’optimisation des performances, de sécurité et de design. Ces améliorations font partie de ma démarche continue d'apprentissage et de perfectionnement. Ce projet incarne ainsi ma passion pour le développement web et ma volonté de contribuer à des projets ayant un impact réel, tout en cherchant constamment à progresser et à améliorer la qualité de mon travail.

Compétences du référentiel couvertes par le projet

1. Développer la partie frontend d’une application web ou web mobile sécurisée

➢ Installer et configurer son environnement de travail en fonction du projet web ou web mobile

➢ Maquetter des interfaces utilisateur web ou web mobile

➢ Réaliser des interfaces utilisateur statiques web ou web mobile

➢ Développer la partie dynamique des interfaces utilisateur web ou web mobile

1. Développer la partie backend d’une application web ou web mobile sécurisée

➢ Mettre en place une base de données relationnelle

➢ Développer des composants d’accès aux données SQL et NoSQL

➢ Développer des composants métier côté serveur

➢ Documenter le déploiement d’une application web ou web mobile

Présentation du projet

Le projet **Arcadia** est une application web et mobile développée pour le parc animalier Arcadia, situé en Bretagne. L'objectif principal de cette application est d'améliorer l'expérience des visiteurs tout en respectant les valeurs écologiques du parc. Le parc souhaite offrir une interface moderne permettant aux visiteurs de découvrir les différents habitats et animaux, ainsi que d'accéder à des services et des informations pratiques comme les horaires, les avis des visiteurs, et un espace de contact.

L'application permet également de gérer les espaces internes du parc grâce à des interfaces dédiées aux administrateurs, employés et vétérinaires. Chaque rôle possède des fonctionnalités spécifiques, telles que la gestion des animaux, des habitats, des services et des avis laissés par les visiteurs. Le projet comprend des éléments avancés de sécurité pour assurer la protection des données des utilisateurs et des systèmes internes. Avec une approche centrée sur l'utilisateur, le projet vise à offrir une interface intuitive et responsive, tout en garantissant la sécurité des données et l'efficacité des opérations internes du parc

Cahier des charges

Le cahier des charges du projet **Arcadia** détaille les besoins fonctionnels et techniques de l'application web et mobile développée pour le parc animalier Arcadia. L'objectif principal est de créer une solution qui permet à la fois d'améliorer l'expérience des visiteurs et de faciliter la gestion interne du parc.

**1. Objectifs du projet**

* Fournir aux visiteurs du parc une application web et mobile intuitive permettant de découvrir les animaux, leurs habitats, ainsi que les services offerts par le parc.
* Offrir des fonctionnalités de gestion adaptées pour les administrateurs, employés et vétérinaires, afin de gérer efficacement les services, les avis, les animaux et leurs états de santé.
* Assurer une expérience utilisateur fluide, interactive et sécurisée pour les visiteurs comme pour le personnel du parc.

**2. Fonctionnalités principales**

* **Page d’accueil** : Présenter le parc, ses valeurs écologiques et un aperçu des animaux et habitats.
* **Menu de navigation** : Fournir un accès rapide aux différentes sections de l'application (Accueil, Services, Habitats, Avis, Connexion).
* **Découverte des habitats et des animaux** : Afficher des informations détaillées sur les animaux du parc et leurs états de santé.
* **Services** : Permettre aux visiteurs de consulter les services offerts (restauration, visites guidées, etc.).
* **Avis des visiteurs** : Offrir la possibilité de laisser des commentaires sur l’expérience au parc, soumis à validation par les employés.
* **Espace Administrateur** : Gérer les comptes utilisateurs, les services, les habitats, et consulter les rapports vétérinaires.
* **Espace Employé** : Valider les avis des visiteurs et gérer les services.
* **Espace Vétérinaire** : Consulter et mettre à jour les informations sur la santé des animaux.
* **Statistiques** : Suivi des consultations et statistiques d'utilisation des différentes sections de l'application.
* **Connexion sécurisée** : Accès restreint pour les administrateurs, employés et vétérinaires via des identifiants sécurisés.

**3. Contraintes techniques**

* **Compatibilité** : L'application doit être accessible sur desktop, tablette et mobile.
* **Sécurité** : Protection des données sensibles avec un système de hachage des mots de passe, protection contre les attaques CSRF et respect des règles du RGPD.
* **Performance** : Optimisation des temps de chargement des pages et gestion efficace des ressources serveur.
* **Base de données** : Utilisation d'une architecture hybride avec une base de données relationnelle (MySQL) pour la gestion des utilisateurs et services, et une base NoSQL (MongoDB) pour la gestion des données non structurées (avis, consultations).
* **Hébergement** : L'application doit être hébergée sur un serveur VPS chez Hostinger, avec une configuration basée sur Ubuntu 22.04 et CloudPanel pour la gestion des ressources serveur.

**4. Technologies utilisées**

* **Front-end** : HTML5, CSS3, JavaScript, Bootstrap, et Twig pour la structure et le rendu des pages.
* **Back-end** : PHP 8.3 avec le framework Symfony, et Doctrine ORM pour la gestion des bases de données relationnelles.
* **Base de données** : MySQL pour la base relationnelle et MongoDB pour la base non relationnelle.
* **API de mailing** : Intégration de Mailjet pour l’envoi de mails automatiques.
* **Gestion de projet** : Utilisation de Trello pour l'organisation des tâches et suivi du projet.

**5. Livrables**

* Code source de l'application (disponible sur GitHub).
* Documentation technique et fonctionnelle.
* Maquettes et wireframes (disponibles sur Figma).
* Application déployée en production avec un accès en ligne sécurisé (lien fourni).

**6. Critères de réussite**

* L’application doit être entièrement fonctionnelle et accessible sur tous les supports.
* Les utilisateurs internes (administrateurs, employés, vétérinaires) doivent pouvoir gérer leurs tâches respectives de manière autonome et sécurisée.
* Les visiteurs doivent pouvoir consulter les informations sur les animaux, les services, et laisser des avis sans difficultés.
* Respect des délais de livraison et des standards de sécurité.

Contraintes

Le projet **Arcadia** doit répondre à plusieurs contraintes, à la fois techniques et organisationnelles, pour assurer son bon fonctionnement et la satisfaction des utilisateurs finaux. Ces contraintes impactent la conception, le développement et la gestion de l'application web et mobile.

**1. Contraintes fonctionnelles**

* **Compatibilité multi-plateforme** : L’application doit être accessible et fonctionner de manière optimale sur différents supports (desktop, tablette, et mobile), avec une interface responsive.
* **Facilité d’utilisation** : L'interface doit être intuitive aussi bien pour les visiteurs que pour le personnel du parc (administrateurs, employés et vétérinaires), en minimisant le nombre de clics pour accéder aux informations.
* **Accès sécurisé** : Les espaces réservés au personnel du parc (administrateurs, employés et vétérinaires) doivent être protégés par une authentification sécurisée avec gestion des rôles utilisateurs.
* **Mise à jour des informations** : Les administrateurs et employés doivent pouvoir mettre à jour facilement les services, habitats, et les états de santé des animaux.

**2. Contraintes de sécurité**

* **Protection contre les intrusions** : Toutes les routes sensibles (accès administrateurs, employés et vétérinaires) doivent être protégées par un système de firewall et de gestion des accès (security.yaml de Symfony).
* **Hachage des mots de passe** : Les mots de passe des utilisateurs doivent être hachés de manière sécurisée avant d'être stockés dans la base de données, en utilisant un algorithme robuste.
* **Prévention des attaques XSS et CSRF** : Le système doit prévenir les attaques par injection de scripts malveillants (XSS) et les attaques par falsification de requêtes intersites (CSRF) via des tokens de protection.

Technologies, outils, Framework

1. Conception et organisation

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. Technologie Frontend

Une image contenant texte, logo, capture d’écran, Police

Description générée automatiquement

1. Backend

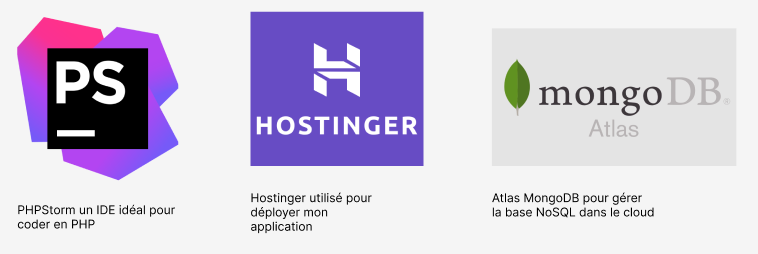
Une image contenant texte, capture d’écran, Police, logo

Description générée automatiquement

1. Bases de données



1. Développement, tests et déploiement



Gestion de projet

Pour organiser le développement de mon application web, j’ai structuré ma gestion de projet sous la forme d’un tableau Kanban avec l’outil Trello. Cela me permet d’avoir une vue globale des différentes étapes et fonctionnalités à développer. Voici comment je me suis organisée.

* 1. Création d’un tableau Trello

J’ai créé un compte Trello, puis un nouveau tableau pour mon projet.

* 1. Ajout des listes

Pour organiser efficacement les tâches de mon projet **Arcadia**, j'ai mis en place un tableau Kanban sur Trello, structuré autour de plusieurs listes. Chaque liste correspond à un état d'avancement des tâches et permet de visualiser le progrès de manière claire et fluide.

* **Idée** : Cette liste contient les idées et les fonctionnalités proposées qui peuvent être envisagées pour le projet, mais qui n'ont pas encore été planifiées.
* **À faire** : Les tâches qui ont été validées pour le développement sont placées dans cette liste. Cela inclut les fonctionnalités front-end et back-end à implémenter.
* **En cours** : Les tâches en cours de développement sont déplacées ici. Cela permet de suivre les activités actives à un moment donné.
* **À déployer en pré-production** : Les tâches qui ont été développées mais qui nécessitent encore des tests ou une validation finale avant d'être déployées en production.
* **Terminer** : Cette liste contient toutes les tâches finalisées et validées. Elle permet de suivre les fonctionnalités qui sont prêtes à être utilisées en production.

Ce tableau m’aide à visualiser rapidement l’état d’avancement du projet, à prioriser les tâches, et à assurer une bonne communication entre les membres de l’équipe. Il facilite également la gestion du temps et des ressources en identifiant les points de blocage et les étapes clés du développement

*Trello présent en annexe (pages 58, 59)*

Conception

1. Diagramme de cas d’utilisation

Le diagramme de cas d'utilisation ci-dessus illustre les différentes interactions possibles entre les utilisateurs du système **Arcadia** et l'application web. Il représente les rôles spécifiques de chaque utilisateur et les fonctionnalités auxquelles ils ont accès.

**1. Admin**

* L’administrateur dispose du contrôle le plus étendu sur l’application. Il peut :
  + Ajouter, modifier ou supprimer des comptes employés et vétérinaires.
  + Consulter le nombre de vues par animal.
  + Gérer les services proposés par le parc, les avis des visiteurs, ainsi que les habitats et les animaux.
  + Accéder aux statistiques de consultation par animal et superviser l’activité générale de l’application.
  + L’administrateur a également accès à la gestion des rapports vétérinaires et aux informations critiques du parc.

**2. Vétérinaire**

* Le vétérinaire a un accès dédié pour consulter et modifier les informations relatives aux animaux. Il peut :
  + Ajouter ou modifier l'alimentation des animaux.
  + Gérer les habitats et le suivi médical des animaux.
  + Rédiger des rapports de santé sur les animaux et transmettre ces informations à l'administration.

**3. Employé**

* L'employé a un rôle de gestion opérationnelle et peut :
  + Ajouter ou modifier l'alimentation des animaux.
  + Gérer les services proposés aux visiteurs, tels que les boutiques ou la restauration.
  + Répondre aux demandes de contact laissées par les visiteurs.
  + Valider ou rejeter les avis déposés par les visiteurs.

**4. Visiteur**

* Le visiteur de l’application a accès à des fonctionnalités publiques. Il peut :
  + Voir les animaux et les consulter par espèce ou par habitat.
  + Déposer un avis concernant sa visite au parc, avis qui sera validé par un employé.
  + Envoyer des demandes de contact à l'administration via un formulaire en ligne.

Le diagramme montre clairement comment chaque type d’utilisateur interagit avec le système. Il permet de visualiser les différentes actions disponibles selon les rôles et de structurer les fonctionnalités du projet en fonction des besoins de chaque utilisateur.

*Diagramme présent en annexe (fig60, 61)*

1. Diagramme de séquence : Connexion de l’administrateur

Le diagramme de séquence ci-dessus montre les différentes interactions qui se produisent lors de la tentative de connexion d'un administrateur à l'application **Arcadia**. Il illustre le flux d’informations entre le front-end, le back-end, et la base de données (BDD).

**1. Étapes du processus**

* **Ouverture de la page de connexion** : L'administrateur commence par accéder à la page de connexion via le front-end de l'application.
* **Affichage du formulaire de connexion** : Le front-end affiche un formulaire de connexion où l'administrateur entre ses identifiants (nom d'utilisateur et mot de passe).
* **Soumission du formulaire** : Une fois que l’administrateur remplit le formulaire, il est soumis au back-end pour vérification.

**2. Vérification des informations**

* **Back-end** : Le back-end reçoit les informations et lance le processus de connexion. Il interroge la base de données pour vérifier si les identifiants fournis sont corrects.
* **Base de données (BDD)** : La BDD renvoie les informations nécessaires au back-end pour vérifier les identifiants de l’administrateur.

**3. Résultats de la vérification**

* **Informations correctes (Data verified)** : Si les informations fournies sont correctes, un message de confirmation est envoyé à l'administrateur via le front-end pour l'informer qu'il est connecté avec succès.
* **Informations incorrectes (Data not verified)** : Si les informations sont incorrectes, un message d’erreur est renvoyé via le front-end, informant l'administrateur que la connexion a échoué.

**4. Condition alternative (Alt)**

* Le diagramme montre également une condition alternative, où en fonction de la vérification des informations (correctes ou non), le système affiche soit un message de confirmation (si la connexion est réussie), soit un message d’erreur (en cas d’échec).

Ce diagramme de séquence permet de visualiser le déroulement logique des interactions nécessaires pour la connexion d'un administrateur, de la soumission du formulaire à la validation des informations via le back-end et la base de données. Il assure ainsi une compréhension claire des différentes étapes et de la gestion des erreurs possibles lors de la connexion.

*Une image contenant texte, capture d’écran, diagramme, Parallèle

Description générée automatiquement*

1. Méthode Merise : Modèle Conceptuel des données (MCD)

Le diagramme représente le **Modèle Conceptuel des Données (MCD)** du projet **Arcadia** selon la méthode Merise. Ce modèle permet de visualiser les entités principales du projet ainsi que les relations entre elles. Il est essentiel pour comprendre la structure de la base de données et les interactions possibles entre les différentes entités.

**1. Entités principales**

* **Service** : Représente les différents services proposés par le parc (boutique, restauration, etc.). Chaque service est identifié par un id unique et possède un nom, une image et une description.
* **Habitat** : Représente les habitats des animaux. Chaque habitat est décrit par son id, un nom, une description, des images et des commentaires. Un habitat peut contenir plusieurs animaux, créant une relation de **contenance** avec l'entité **Animal**.
* **Animal** : Cette entité représente les animaux du parc, identifiés par un id, un nom, une race et des images. Chaque animal est lié à un ou plusieurs habitats et reçoit de la nourriture, ce qui crée une relation avec l'entité **Animal\_feeding**.
* **Animal\_feeding** : Cette entité gère les informations sur l'alimentation des animaux, avec les détails de la date de nourrissage, le type de nourriture et la quantité.
* **Review** : Représente les avis laissés par les visiteurs, avec un id, le pseudo du visiteur, l'avis proprement dit, un statut de validation, la date de soumission et une note.
* **Veterinary\_report** : Cette entité regroupe les rapports des vétérinaires sur l’état de santé des animaux, avec des informations comme l'état de santé, la date du rapport, les détails sur la nourriture, et le poids de l'animal.
* **Users** : Gère les informations des utilisateurs de l’application (employés, vétérinaires, administrateurs). Chaque utilisateur est identifié par un id, un email, un mot de passe, un rôle (admin, employé, vétérinaire), et un token pour réinitialiser le mot de passe.
* **Contact** : Représente les demandes de contact envoyées par les visiteurs, incluant un id, un titre, une description et une adresse email.
* **Schedule** : Gère les horaires d’ouverture et fermeture du parc. Cette entité inclut les jours d'ouverture/fermeture et les horaires associés.

**2. Relations entre les entités**

* **Contenir (0,n : 1,1)** : Un habitat peut contenir plusieurs animaux, mais chaque animal appartient à un seul habitat.
* **Appartiens (1,n : 1,1)** : Un animal peut appartenir à plusieurs habitats, et chaque habitat peut avoir plusieurs animaux, créant une relation bidirectionnelle.
* **Mange (0,n : 1,1)** : Un animal a plusieurs enregistrements de nourrissage, mais chaque enregistrement de nourrissage est lié à un seul animal.
* **Donne (1,1 : 0,n)** : Chaque rapport vétérinaire est rédigé par un utilisateur (vétérinaire), et un vétérinaire peut rédiger plusieurs rapports.

**3. Objectifs du MCD**

Ce modèle conceptuel permet de définir la structure des données qui seront utilisées dans l’application. Il assure une gestion efficace des relations entre les différentes entités et optimise l'organisation des informations dans la base de données. Le modèle Merise permet également de mieux comprendre les interactions entre les différentes composantes du parc (animaux, habitats, services, utilisateurs) et d’assurer la cohérence des informations stockées.

Une image contenant texte, diagramme, capture d’écran, Plan

Description générée automatiquement

Charte graphique

La charte graphique est un document de travail qui contient l’ensemble des règles fondamentales d’une entreprise ; dans mon cas le zoo Arcadia. Ce support constitue l’ensemble de l’identité visuelle avec le logo, la police de caractère et les codes couleurs. Elle est aussi le pilier de la stratégie de communication car elle garantit la cohérence de l’identité visuelle sur tous supports (application web, flyers, carte de visite…). J’ai utilisé l’outil Figma pour la création de cette charte graphique.

Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement

Une image contenant texte, capture d’écran, Rectangle, conception

Description générée automatiquement

Wireframes et maquettes

* 1. Wireframe
* Un **wireframe** est un schéma utilisé pour la conception d’une interface utilisateur dans le but de définir les zones et composants qu’elle doit contenir. Il est schématisé par des blocs afin de déterminer les grandes fonctionnalités. Les wireframes jouent un rôle clé dans la **recherche d'ergonomie** et permettent de tester différentes dispositions avant de finaliser l’interface. Leur objectif principal est d’améliorer l’expérience utilisateur (UX) en garantissant que l'interface est intuitive, fonctionnelle et répond aux attentes des utilisateurs.
* Les **wireframes** permettent ainsi d'optimiser la structure des pages en mettant en avant les éléments les plus importants et en garantissant une navigation fluide. Ils aident également à identifier les zones à améliorer avant de passer aux étapes de conception visuelle.

*Wireframes présents en annexe (pages 63)*

2.Maquettes

* Une **maquette** (ou **Mockup**) est une représentation visuelle et réaliste du futur site web. C’est un modèle statique qui simule l’apparence finale de l’interface utilisateur avant le processus de développement. Contrairement aux wireframes, les maquettes intègrent tous les éléments graphiques : la charte graphique, les couleurs, la typographie, les logos, les titres, et les icônes.
* Les maquettes sont essentielles pour visualiser la hiérarchie visuelle et s’assurer que le design est en adéquation avec les objectifs du projet. Elles permettent également de vérifier la cohérence visuelle et d’évaluer l'impact des choix graphiques sur l’**expérience utilisateur**. En intégrant la charte graphique et les éléments visuels, elles aident à valider l'ergonomie tout en reflétant fidèlement l’identité visuelle de l'application.

*Maquettes présentes en annexe (pages 64)*

Installation et configuration de l’environnement de travail

*OS utilisé : Windows 10*

1. Installation de l’IDE

Pour le développement de mon projet **Arcadia**, j’ai choisi d’utiliser **PHPStorm** comme environnement de développement intégré (IDE). Ce choix s'explique par les nombreuses fonctionnalités avancées que cet IDE offre pour le développement en PHP et en Symfony.

**Pourquoi PHPStorm ?**

PHPStorm est particulièrement adapté aux projets en **PHP** et **Symfony**, grâce à ses capacités de gestion de grands projets, son débogueur intégré, et ses outils d'analyse de code. Voici quelques raisons spécifiques pour lesquelles j’ai opté pour PHPStorm :

* **Support des principaux langages** : PHPStorm prend en charge PHP, JavaScript, HTML, CSS, ainsi que d’autres langages couramment utilisés dans le développement web.
* **Extensions et plugins** : Il propose une large gamme de plugins et d’extensions pour personnaliser l’environnement selon mes besoins. Par exemple, des plugins spécifiques à Symfony et Twig facilitent la gestion des templates.
* **Outils intégrés** : PHPStorm intègre de nombreux outils de développement tels que :
  + **Terminal intégré** : Permet d’exécuter des commandes directement depuis l’IDE sans avoir à ouvrir un terminal externe.
  + **Exécution du code et débogage** : Avec un débogueur avancé et une prise en charge des tests unitaires, PHPStorm simplifie le processus de développement et de correction des erreurs.
  + **Contrôle de version** : PHPStorm intègre nativement **Git** et **GitHub**, facilitant la gestion des versions du projet et le travail collaboratif.
* **Analyse statique** : PHPStorm propose une analyse statique qui permet d'identifier les erreurs potentielles avant même d'exécuter le code.

**Installation de PHPStorm**

L’installation de PHPStorm est simple. J’ai téléchargé l’IDE directement depuis le site officiel :  
<https://www.jetbrains.com/phpstorm/>

J'ai ensuite suivi la documentation officielle disponible à cette adresse pour configurer mon environnement :  
<https://www.jetbrains.com/phpstorm/documentation/>

Grâce à ces riches fonctionnalités et à la prise en charge complète de Symfony, PHPStorm s'est avéré être un outil précieux pour développer efficacement et en toute sécurité l'application **Arcadia**.

1. Symfony

J’ai choisi de travailler avec le framework web open source **Symfony**, qui suit également le principe **Modèle - Vue - Contrôleur (MVC)**. Symfony est reconnu pour sa robustesse, sa flexibilité, et sa capacité à créer des applications web scalables et performantes.

**Pourquoi Symfony ?**

Le choix de Symfony pour le projet **Arcadia** s'explique par les avantages suivants :

* **Architecture MVC** : Symfony organise le code en séparant clairement la logique de présentation (Vue), la logique métier (Modèle), et les interactions utilisateur (Contrôleur), facilitant ainsi la maintenance et l’évolutivité de l'application.
* **Composants modulaires** : Symfony offre un ensemble de composants réutilisables qui facilitent le développement de fonctionnalités complexes tout en garantissant une architecture claire et modulaire.
* **Sécurité intégrée** : Symfony propose des outils intégrés pour la gestion de la sécurité (authentification, autorisation, protection contre les attaques CSRF), ce qui est un aspect clé du projet Arcadia.

**Installation de Symfony**

Pour installer Symfony, je me suis rendu sur le site officiel et j’ai suivi la documentation disponible à cette adresse :  
https://symfony.com/doc/current/index.html

Dans le terminal, j'ai utilisé la commande suivante pour créer le projet **Arcadia** :

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

Cette commande génère la structure de base du projet, avec tous les fichiers nécessaires pour commencer le développement.

**Démarrage du serveur**

Pour démarrer le serveur local et tester l’application en développement, j’utilise la commande suivante dans le terminal :

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

Cette commande démarre un serveur local sur lequel l’application est accessible via le navigateur, ce qui permet de tester et de voir les changements en temps réel pendant le développement.

**Conclusion**

Symfony est un framework puissant et adapté aux projets web complexes. Son architecture MVC, ses composants flexibles et sa documentation complète en font un outil de choix pour le développement de l'application **Arcadia**. Grâce à ses fonctionnalités intégrées, telles que la gestion de la sécurité et la manipulation des bases de données via Doctrine, Symfony facilite le développement d'une application robuste, évolutive et sécurisée.

1. Git pour GitHub

GitHub est une plateforme de développement qui permet de créer, stocker, gérer et de partager son code avec d’autres développeurs.

Concernant l’installation, j’ai tout d’abord créé un nouveau « repository » depuis l’adresse : https://github.com/ .

J’ai ensuite initié mon projet depuis le terminal de VS Code en utilisant les commandes suivantes (commandes suivies depuis la documentation) :

➢ git init

➢ git add .

➢ git commit -m « first commit »

➢ git push -u origin main

Une fois le « commit » initial réaliser, j’ai ajouté une branche. Pour cela depuis mon terminal j’ai utilisé les commandes suivantes :

➢ git checkout -b develop

➢ git branch

➢ git push origin develop

5. MySQL

MySQL est un système de gestion de données relationnelles (SGBDR) qui permet de créer et administrer une base de données dans lequel on peut effectuer des requêtes SQL.

1. MongoDB

MongoDB est un système de base de données orienté documents et ne nécessitant pas de schéma prédéfini des données.

MongoDB sera utilisé dans mon cas dans l’incrémentation des consultations animaux.

Pour utiliser MongoDB, j’ai opté pour le choix de l’hébergement cloud que propose MongoDB : MongoDB ATLAS

Pour l’utiliser, je me suis créée un compte puis un projet et j’ai copié les informations de connexion à ma base de données dans mon projet.

1. Bootstrap

Pour styliser l'application **Arcadia**, j'ai choisi d'utiliser **Bootstrap** via un CDN (Content Delivery Network). L'utilisation de Bootstrap via CDN permet de charger rapidement les styles CSS et les scripts JavaScript sans avoir à installer et configurer le framework localement. Cette approche est simple et pratique pour garantir que l'application bénéficie toujours des dernières mises à jour de Bootstrap.

**Pourquoi Bootstrap via CDN ?**

L'utilisation de Bootstrap via un CDN présente plusieurs avantages :

* **Rapidité de chargement** : Les fichiers sont hébergés sur des serveurs distribués dans le monde entier, ce qui réduit le temps de chargement des pages pour les utilisateurs.
* **Mise à jour automatique** : En utilisant un CDN, l'application profite des dernières versions de Bootstrap sans avoir besoin de les mettre à jour manuellement.
* **Simplicité d'intégration** : Il suffit d'ajouter quelques lignes de code HTML pour intégrer Bootstrap dans l'application, ce qui accélère le processus de développement.

**Intégration de Bootstrap via CDN**

J’ai suivi la documentation officielle sur ce lien : <https://getbootstrap.com/docs/5.0/getting-started/introduction/>

Développement

* 1. Twig

Dans le cadre du développement de l'application **Arcadia**, j'ai choisi d'utiliser **Twig**, le moteur de templates pour **Symfony**, afin de générer les vues dynamiques de l'application. Twig est un outil puissant et flexible qui permet de séparer la logique de présentation de la logique métier, garantissant ainsi un code plus propre et maintenable.

**Pourquoi Twig ?**

Twig présente plusieurs avantages :

* **Séparation claire des responsabilités** : Grâce à Twig, le code HTML de l'application est séparé de la logique métier, ce qui facilite la maintenance et améliore la lisibilité du code.
* **Syntaxe simple et puissante** : La syntaxe de Twig est facile à comprendre et à utiliser, tout en offrant des fonctionnalités avancées comme les boucles, les conditions, et l'héritage de templates.
* **Sécurité intégrée** : Twig intègre des fonctions de filtrage automatique qui aident à prévenir les failles XSS (Cross-Site Scripting), ce qui est essentiel pour garantir la sécurité des données affichées à l'utilisateur.
* **Réutilisabilité des templates** : Twig permet de créer des blocs de templates réutilisables, ce qui réduit la duplication de code et améliore la gestion des vues.
* **Utilisation des templates Twig** : Dans les fichiers de l’application, les vues sont créées avec des templates Twig (.twig). Ces fichiers contiennent le code HTML, avec des balises Twig pour intégrer des éléments dynamiques, comme les variables, les boucles, et les conditions

**Exemple d’utilisation de Twig**

Voici un exemple simple d’utilisation de Twig pour afficher des données dynamiques dans une vue :

1. **Héritage de template** : J’utilise souvent un fichier de base pour l'ensemble des pages de l’application, puis je fais hériter d'autres templates pour garder une structure commune :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

1. **Utilisation dans une page** : Un autre fichier Twig héritera de ce template de base et remplira les blocs de contenu spécifiques :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

**Avantages de Twig pour le projet Arcadia**

1. **Réutilisation des templates** : J'ai pu réutiliser des éléments communs comme les en-têtes, les pieds de page et les barres de navigation dans plusieurs pages en utilisant l'héritage de templates. Cela m’a permis de maintenir une interface cohérente et de réduire la duplication de code.
2. **Gestion des données dynamiques** : Grâce à Twig, j’ai facilement affiché des données provenant de la base de données, comme la liste des animaux et des habitats, dans des boucles dynamiques, tout en filtrant les données pour garantir leur sécurité.
3. **Simplicité de gestion des formulaires** : Twig facilite également l'affichage et la gestion des formulaires en utilisant des fonctions spécifiques de Symfony, ce qui a simplifié le développement des fonctionnalités de connexion, d’avis, et de gestion des services du parc.

**Conclusion**

L'utilisation de **Twig** dans le projet **Arcadia** m’a permis de structurer proprement les vues, de réutiliser des blocs de code, et de gérer facilement l'affichage dynamique des données. Ce moteur de templates s'est révélé être un excellent choix pour maintenir un code clair, sécurisé et modulable tout au long du développement de l'application.

2. Le style avec Bootstrap

Pour la conception de l'interface utilisateur du projet **Arcadia**, j'ai utilisé **Bootstrap**, un framework CSS populaire qui facilite la création d'interfaces modernes, réactives et esthétiques. Grâce à ses classes utilitaires et composants préconstruits, Bootstrap m’a permis de mettre en place un design harmonieux et cohérent tout en assurant une excellente expérience utilisateur.

**Pourquoi Bootstrap ?**

J'ai choisi **Bootstrap** pour plusieurs raisons :

* **Responsive design** : Bootstrap est conçu pour être mobile-first, ce qui garantit que l'application s’adapte automatiquement à toutes les tailles d’écran (smartphone, tablette, desktop).
* **Grille CSS flexible** : Le système de grille 12 colonnes de Bootstrap m’a permis d'organiser facilement les éléments sur les pages, créant ainsi une interface bien structurée et lisible sur tous les appareils.
* **Composants préconstruits** : Bootstrap propose une variété de composants (boutons, formulaires, modales, cartes) qui m’ont permis de créer des interfaces interactives rapidement.
* **Personnalisation facile** : Bien que Bootstrap offre des styles par défaut, il permet également une personnalisation poussée des couleurs, des typographies, et d'autres aspects du design, pour s’adapter aux besoins spécifiques de chaque projet.

**Utilisation des classes Bootstrap**

J’ai principalement utilisé les classes utilitaires et les composants de Bootstrap pour structurer et styliser les pages du projet. Voici quelques exemples concrets d’utilisation de Bootstrap dans **Arcadia** :

1. **Grille responsive** : Pour organiser le contenu des pages, j’ai utilisé le système de grille de Bootstrap, ce qui a permis de garantir une structure responsive. Et grâce à Twig je peux effectuer des boucles, des conditions et afficher directement des données.
2. **Composants** : J'ai également utilisé divers composants Bootstrap tels que les modales et les cartes pour offrir une expérience utilisateur interactive.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. Javascript dans mes pages dynamiques

Pour rendre certaines parties de l’application plus interactives et dynamiques, j’ai intégré des fonctionnalités JavaScript. JavaScript a été utilisé pour manipuler le DOM (Document Object Model), permettant d’ajouter des interactions utilisateur telles que les modales, les validations de formulaires en temps réel, ou encore la gestion dynamique de contenu sans rechargement de page. Grâce à JavaScript, les pages bénéficient d'une expérience utilisateur enrichie et plus fluide.

1. Requête HTTP et fonction asynchrone

Pour améliorer la réactivité de l'application, j’ai utilisé des requêtes HTTP asynchrones via AJAX et la fonction fetch en JavaScript. Cela a permis de charger ou de mettre à jour du contenu sur la page sans nécessiter de rechargement complet, notamment pour les soumissions de formulaires ou la récupération de données. Cette approche améliore l'efficacité et la rapidité de l'application, en particulier pour les interactions fréquentes avec le serveur.

a. DOMContentLoaded Event Listener:



**But** : Cette ligne écoute l'événement **DOMContentLoaded**, qui est déclenché lorsque le HTML de la page est complètement chargé et parsé par le navigateur.

**Pourquoi** : Cela garantit que tous les éléments de la page sont accessibles via le DOM avant que le code JavaScript ne commence à les manipuler.

b. Sélection de tous les liens d'habitat :



**But** : Ici, **querySelectorAll** sélectionne tous les éléments avec la classe **habitat-link** (supposément des liens HTML) et les parcourt avec **forEach**.

**Pourquoi** : Le but est d’ajouter un événement click à chacun de ces liens, afin de déclencher une action lorsqu'un lien est cliqué.

c. Ajout d'un événement click :



**But** : Cette partie du code attache un **écouteur d'événements** click à chaque lien d'habitat. Lorsque le lien est cliqué, la fonction associée est exécutée.

**event.preventDefault()** : Cette méthode empêche le comportement par défaut du lien (redirection vers une nouvelle page). Cela permet de gérer l'événement en JavaScript sans rechargement de page.

d. Récupération de l'identifiant de l'habitat :



**But** : Ici, on récupère l'attribut personnalisé data-habitat-id du lien cliqué. Cet attribut est utilisé pour identifier l'habitat spécifique sélectionné.

**Pourquoi** : L’**id** de l'habitat sera utilisé pour envoyer une requête au serveur et obtenir des informations sur cet habitat et les animaux associés.

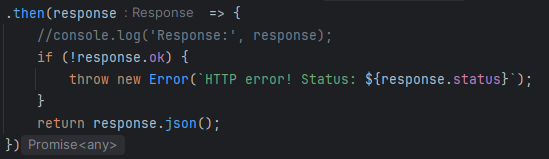
e. Envoi de la requête fetch :



**But** : La méthode **fetch** est utilisée pour envoyer une requête HTTP **GET** asynchrone au serveur. Elle interroge l'URL /habitat/${habitatId}/animals, où ${habitatId} est l'identifiant de l'habitat que nous avons récupéré précédemment.

**Pourquoi** : Le serveur va répondre avec les détails sur l'habitat et les animaux sous forme de JSON.

f. Vérification de la réponse :



**But** : Après avoir reçu la réponse du serveur, on vérifie si la réponse est correcte avec response.ok. Si le statut de la réponse n'est pas OK (par exemple, 404 ou 500), une erreur est levée.

**Pourquoi** : Cela permet de gérer les erreurs de manière propre. Si la réponse est valide, elle est convertie en **JSON** via response.json() pour faciliter son utilisation dans le reste du script.

g. Mise à jour dynamique du contenu avec les données reçues :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**But** : Après avoir reçu et converti les données en JSON, on sélectionne l’élément avec l’ID **animals-list**, qui représente probablement la zone où les informations sur les animaux vont être affichées. Ensuite, on vide cette zone avec innerHTML = '' pour s'assurer qu'elle est prête à recevoir les nouvelles données.

**Pourquoi** : On veut garantir que la liste soit réinitialisée à chaque clic, pour que seules les nouvelles données soient affichées.

h. Affichage de la description de l'habitat :

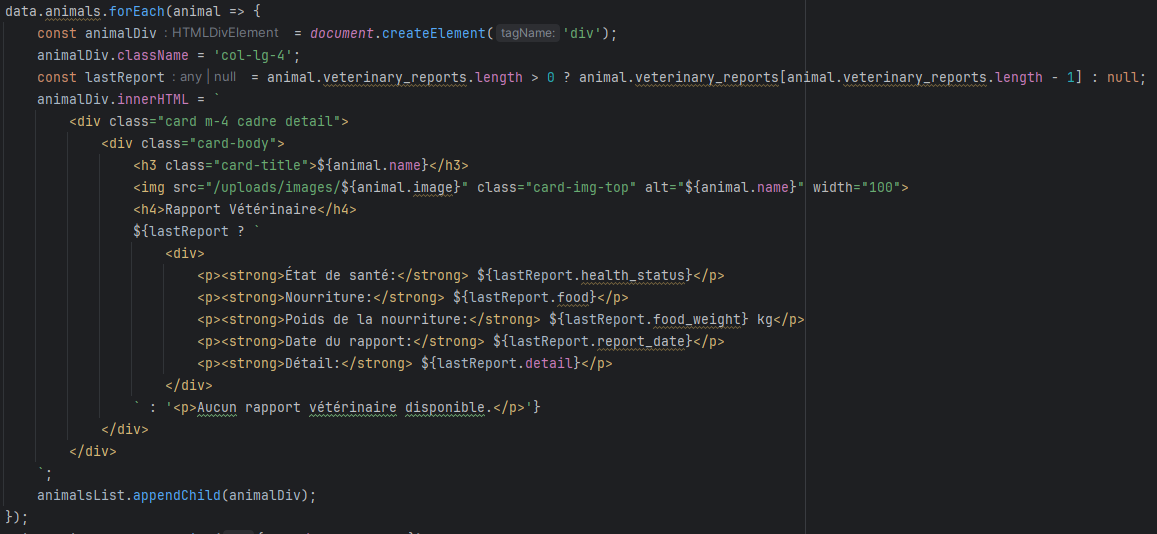
Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

**But** : Ici, une nouvelle div est créée pour contenir les détails de l'habitat. Elle contient des informations telles que le nom et la description de l'habitat (récupérées via data.habitat\_name et data.habitat\_detail).

**Pourquoi** : Ces données sont affichées dans un style de carte Bootstrap pour présenter les informations à l'utilisateur dans un format esthétique.

i. Affichage des animaux dans l'habitat :



**But** : Ici, pour chaque animal récupéré, une nouvelle div est créée et remplie avec les détails de l'animal (nom, image, dernier rapport vétérinaire, etc.). Si l'animal a un rapport vétérinaire, celui-ci est également affiché ; sinon, un message indiquant l'absence de rapport est montré.

**Pourquoi** : Ces données sont récupérées directement depuis la réponse JSON et affichées de manière dynamique dans le DOM, offrant une interaction enrichie pour l'utilisateur.

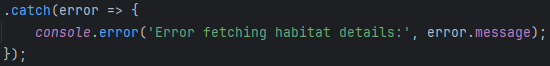
j. Défilement fluide :



**But** : Cette ligne assure un défilement fluide vers la section des détails des animaux lorsque les données sont chargées.

**Pourquoi** : Cela améliore l'expérience utilisateur en garantissant que l'utilisateur est directement amené à l'endroit où les informations sont affichées.

k. Gestion des erreurs :



**But** : En cas d'erreur lors de la requête ou du traitement de la réponse, cette section capture l'erreur et affiche un message d'erreur dans la console.

**Pourquoi** : Cela permet de gérer les erreurs de manière propre et de fournir des informations utiles pour le débogage.

**Conclusion**

Ce code permet de rendre une page web interactive et dynamique en récupérant des informations à partir d'un serveur à l'aide de requêtes **HTTP asynchrones** via **fetch**, puis en manipulant le DOM pour afficher ces informations sans recharger la page. Cela améliore considérablement l'expérience utilisateur en rendant l'application plus fluide et réactive.

1. Base de données et ORM Doctrine

Dans le cadre du projet **Arcadia**, j’ai utilisé **Doctrine ORM** pour la gestion de la base de données. Doctrine est un **ORM (Object-Relational Mapping)** qui facilite l’interaction avec la base de données en transformant les entités PHP en tables SQL et vice-versa. Cela permet de manipuler les données sous forme d’objets dans le code, sans avoir à écrire manuellement des requêtes SQL complexes.

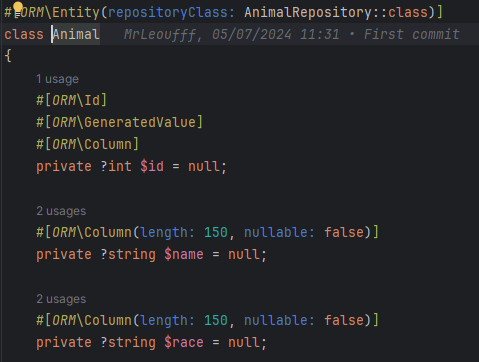
**Pourquoi utiliser Doctrine ?**

* **Abstraction des requêtes SQL** : Avec Doctrine, il n'est pas nécessaire d'écrire des requêtes SQL manuellement. Doctrine génère automatiquement les requêtes SQL pour la création, la lecture, la mise à jour et la suppression (CRUD) des données.
* **Mapping objet-relationnel** : Doctrine permet de mapper les entités PHP aux tables de la base de données, ce qui facilite la gestion des relations entre les entités (one-to-one, one-to-many, many-to-many).
* **Gestion des migrations** : Doctrine intègre un système de migration pour suivre et appliquer les changements dans la structure de la base de données au fur et à mesure du développement.

**Modélisation des entités**

Dans **Arcadia**, les entités principales incluent des entités comme **Habitat**, **Animal**, et **VeterinaryReport**. Chaque entité est associée à une table dans la base de données, et les relations entre les entités sont gérées par Doctrine.

Exemple d’entité : Animal

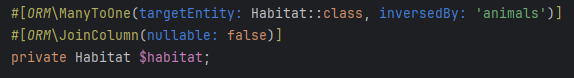


**Annotations Doctrine** : Les annotations comme @ORM\Column et @ORM\ManyToOne permettent à Doctrine de comprendre comment cette entité est mappée à la base de données.

**Relations entre entités** : L’entité Animal est reliée à l’entité Habitat via une relation **ManyToOne** (plusieurs animaux peuvent appartenir à un même habitat). Elle a aussi une relation **OneToMany** avec les rapports vétérinaires (VeterinaryReport), signifiant qu'un animal peut avoir plusieurs rapports vétérinaires.

**Gestion des relations**

Doctrine simplifie la gestion des relations entre les entités. Par exemple, dans l'entité Animal, la relation avec Habitat est définie par @ManyToOne, ce qui signifie qu'un habitat peut avoir plusieurs animaux. De même, la relation entre Animal et VeterinaryReport est définie par @OneToMany, ce qui signifie qu'un animal peut avoir plusieurs rapports vétérinaires.



Exemple d'une requête avec Doctrine

$animals = $entityManager->getRepository(Animal ::class)->findBy([‘habitat’ => $habitatId]) ;

Cette requête récupère tous les animaux appartenant à un habitat spécifique, en utilisant le repository de Doctrine pour interagir avec la base de données.

La même requête sans Doctrine

SELECT \* FROM animal WHERE habitat\_id = :habitatId;

**Explications :**

* **animal** : C'est le nom de la table correspondant à l'entité Animal.
* **habitat\_id** : C'est la clé étrangère qui relie les animaux à leur habitat.
* **:habitatId** : C'est le paramètre qui doit être remplacer par l'identifiant ($habitatId) lors de l'exécution.

Avantages de Doctrine pour le projet Arcadia :

1. **Facilité de gestion des relations** : Doctrine facilite la gestion des relations entre les entités (par exemple, la relation entre Animal et VeterinaryReport), permettant de récupérer et manipuler des données complexes avec des requêtes simples.
2. **Migration facile** : Doctrine offre un système de **migrations** qui permet de suivre et d’appliquer facilement les changements de structure dans la base de données.
3. **Optimisation des requêtes** : Doctrine optimise les requêtes SQL, ce qui permet de réduire la complexité des interactions avec la base de données.

**Conclusion**

Grâce à **Doctrine ORM**, la gestion de la base de données dans **Arcadia** est simplifiée et optimisée. En utilisant une approche orientée objet, Doctrine permet de manipuler les données de manière intuitive tout en bénéficiant d'un puissant système de migration et de gestion des relations entre les entités.

Exemple sans ORM Doctrine :

Pour créer la BDD

CREATE DATABASE IF NOT EXISTS arcadia CHARACTER SET utf8 COLLATE utf8\_general\_ci;

Pour créer une table

USE arcadia

CREATE TABLE IF NOT EXISTS animal (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 name VARCHAR(150) NOT NULL,  
 race VARCHAR(150) NOT NULL,  
 image JSON,  
 habitat\_id INT,  
 FOREIGN KEY (habitat\_id) REFERENCES habitat(id)  
);

Pour insérer des données

INSERT INTO 'animal'  
 ('id', 'habitat\_id', 'name', 'race', 'image')  
VALUES  
 (8, 2, 'Wivine', 'Vélociraptor', '[\"velociraptor-66911c2e52919.jpg\"]'),  
 (11, 12, 'Seb', 'Diplodocus', '[\"diplodocus-6693e96007d6d.jpg\"]'),  
 (12, 2, 'Capt\'', 'Spinosaure', '[\"spinosaurus-6693f36737411.jpg\"]'),  
 (13, 2, 'Exploris', 'T-Rex', '[\"1526479615762038050-669413357050c.jpg\"]');

1. Les Controllers

Dans l'application **Arcadia**, les **Controllers** jouent un rôle central dans le modèle **MVC (Modèle-Vue-Contrôleur)** de Symfony. Les **controllers** sont responsables de la gestion des requêtes des utilisateurs, de la manipulation des données à l'aide des services et des entités, et de la préparation des réponses envoyées aux utilisateurs, qu'il s'agisse de vues HTML, de JSON, ou d'autres formats.

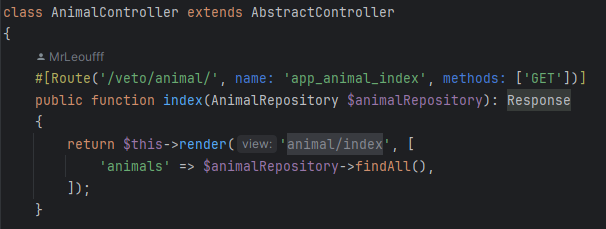
**Qu'est-ce qu'un Controller dans Symfony ?**

Un **Controller** est une classe PHP qui gère la logique des actions à réaliser lorsque l'utilisateur interagit avec l'application. Par exemple, lorsqu'un utilisateur accède à une page spécifique, remplit un formulaire, ou clique sur un lien, le controller correspondant traite la requête, interagit avec le modèle (entités, base de données via Doctrine), et retourne une réponse.

**Exemple de Controller : AnimalController**

Voici un exemple de controller qui gère les animaux dans l'application **Arcadia**. Ce controller récupère les informations sur les animaux depuis la base de données et les affiche à l'utilisateur.

**Route /veto/animal/ - Index des Animaux**



**But** : Cette route retourne une liste de tous les animaux. La méthode utilise le **repository** pour interroger la base de données et récupérer tous les enregistrements d'animaux.

**Rendu** : Les résultats sont envoyés à la vue **index.html.twig**, où la liste des animaux sera affichée.

**Route /veto/animal/new - Création d'un Nouvel Animal**

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

**But** : Cette route permet de créer un nouvel animal via un formulaire. Lorsqu'une demande **GET** est envoyée, le formulaire est affiché. Si une demande **POST** est envoyée (soumission du formulaire), les données sont validées et enregistrées dans la base de données.

**Gestion des images** : Les images des animaux sont traitées, renommées de manière sécurisée, puis stockées dans un répertoire spécifié.

**Redirection** : Si l'animal est créé avec succès, l'utilisateur est redirigé vers la liste des animaux.

*Code complet en annexe(fig)*

**Route /animal - Liste des Animaux**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**But** : Cette route affiche la liste de tous les animaux disponibles dans la base de données.

**Vue** : La liste est transmise au fichier **list.html.twig**, où chaque animal est affiché.

**Route /veto/animal/{id}/edit - Édition d'un Animal**

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

**But** : Cette route permet de modifier les informations d'un animal. Si l'utilisateur soumet un formulaire, les informations de l'animal sont mises à jour, y compris les images.

**Validation** : Le formulaire vérifie la validité des données (comme la sélection de l'habitat) avant de procéder à la mise à jour.

**Route /veto/animal/{id} - Suppression d'un Animal**

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

**But** : Cette route permet de supprimer un animal de la base de données via une requête POST. Le système vérifie la validité du token CSRF avant de procéder à la suppression.

**Confirmation** : Après suppression, l'utilisateur est redirigé vers la liste des animaux avec un message de confirmation.

**Route /animal/{id} - Détails d'un Animal**

Une image contenant texte, capture d’écran

Description générée automatiquement

**But** : Cette route affiche les détails d'un animal particulier, y compris les rapports vétérinaires et les données sur la nourriture.

**Tracking des vues** : Chaque fois qu’un animal est affiché, un compteur de vues est mis à jour à l'aide de MongoDB (DocumentManager) pour enregistrer combien de fois chaque animal a été vu.

**Vue** : Le template **show.html.twig** est utilisé pour afficher les détails de l'animal, ainsi que les rapports et les informations sur son alimentation.

*Code complet en annexe (fig)*

**Conclusion**

Le controller **AnimalController** est bien structuré et gère les opérations CRUD (Create, Read, Update, Delete) pour l'entité **Animal**. Il intègre également la gestion des images et la mise à jour des informations associées, comme les rapports vétérinaires et l'alimentation des animaux. Chaque route est soigneusement configurée pour répondre à des besoins spécifiques et l'utilisation des formulaires Symfony permet une gestion efficace des soumissions et des validations des données.

1. Form Requests dans Symfony

Dans **Symfony**, les **Form Requests** (ou formulaires) permettent de gérer et de valider les données soumises par les utilisateurs via des formulaires HTML. Symfony propose un système de gestion de formulaires puissant, qui inclut des fonctionnalités comme la création de formulaires, la gestion de la validation des données, et le traitement sécurisé des soumissions.

**Pourquoi utiliser les Form Requests ?**

Les **Form Requests** dans Symfony facilitent :

* **La création de formulaires complexes** : En définissant la structure des formulaires de manière déclarative.
* **La validation des données** : En spécifiant les règles de validation dans des classes de formulaire.
* **La sécurité des formulaires** : Grâce à des fonctionnalités comme la protection CSRF et le nettoyage des données.

Le formulaire est intégré dans le **Controller** pour traiter la création ou la modification d'un animal. Par exemple, voici comment le formulaire est utilisé dans la méthode new du **AnimalController** :

Symfony offre un système de validation des données via des **contraintes**. Par exemple, dans **AnimalType**, la contrainte **File** impose des restrictions sur les fichiers téléchargés. Ces contraintes peuvent être étendues pour valider d'autres types de données comme les chaînes de caractères, les dates, etc.

Symfony intègre automatiquement la protection **CSRF (Cross-Site Request Forgery)** pour empêcher les attaques potentielles. Chaque formulaire généré contient un **jeton CSRF**, qui est vérifié lors de la soumission du formulaire. Cette protection est activée par défaut dans les formulaires Symfony, rendant l'application plus sécurisée.

**Conclusion**

Les **Form Requests** dans Symfony permettent de gérer de manière sécurisée, efficace et extensible les interactions entre les utilisateurs et les entités du projet **Arcadia**. Grâce à la validation, aux contraintes, et à l'intégration native de la protection CSRF, Symfony assure une gestion des formulaires robuste et sécurisée, facilitant ainsi la soumission et le traitement des données.

Mailling

**Mailing dans Symfony avec le MailerService**

Dans ce projet, le service **MailerService** gère l'envoi des emails via le composant **Mailer** de Symfony. Il est utilisé pour envoyer des emails de contact ainsi que des emails de notification aux utilisateurs lors de la création de leur compte.

Les Dépendances du Service

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

**$zooEmail** : Il s'agit de l'email de l'organisation (ici le zoo) défini dans la configuration de l'application (via la variable %zoo\_email%).

**$mailer** : Il s'agit de l'instance de **MailerInterface** injectée dans le service pour envoyer les emails. Symfony gère cette interface via le composant **Mailer**.

Le Constructeur du Service

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

**#[Autowire('%zoo\_email%')]** : Cette annotation permet d'injecter automatiquement l'email du zoo à partir du fichier de configuration (services.yaml, par exemple). Cela évite de devoir passer cet email manuellement.

**MailerInterface** : L'instance du composant **Mailer** est injectée ici pour gérer l'envoi des emails.

Méthode sendContactEmail()

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

**from($contact->getEmail())** : L'email est envoyé par l'adresse saisie dans le formulaire de contact.

**to($this->zooEmail)** : L'email est envoyé à l'adresse du zoo.

**htmlTemplate()** : Un modèle **Twig** (fichier emails/contact.html.twig) est utilisé pour le contenu de l'email.

**context()** : Un tableau de données est passé au template pour afficher dynamiquement le contenu dans l'email (titre, description, et email de l'expéditeur).

Méthode sendUserCreationEmail()

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

**from($this->zooEmail)** : L'email du zoo est utilisé comme expéditeur.

**to($user->getEmail())** : L'email est envoyé à l'utilisateur, à l'adresse associée à son compte.

**htmlTemplate()** : Un modèle **Twig** est utilisé pour le contenu HTML de l'email.

**context()** : Ici, on passe l'email de l'utilisateur au modèle Twig pour l'afficher dans le contenu de l'email.  **from($this->zooEmail)** : L'email du zoo est utilisé comme expéditeur.

**to($user->getEmail())** : L'email est envoyé à l'utilisateur, à l'adresse associée à son compte.

**htmlTemplate()** : Un modèle **Twig** est utilisé pour le contenu HTML de l'email.

**context()** : Ici, on passe l'email de l'utilisateur au modèle Twig pour l'afficher dans le contenu de l'email.

Utilisation des Templates Twig pour les Emails

Les méthodes sendContactEmail et sendUserCreationEmail utilisent toutes deux des **modèles Twig** pour générer le contenu HTML des emails.

Exemple du fichier contact.html.twig :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Exemple du fichier user\_creation.html.twig :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Les modèles Twig permettent de séparer la logique de la génération des emails et le contenu. Cela rend l'envoi d'emails plus flexible et maintenable.

**Conclusion**

Le **MailerService** gère de manière centralisée l'envoi d'emails dans le projet **Arcadia**. Il utilise le composant **Mailer** de Symfony, qui s'intègre parfaitement avec Twig pour générer des emails HTML dynamiques. Grâce à l'injection de dépendances et à l'utilisation des templates, ce service permet une gestion simple, flexible et maintenable des communications par email.

Sécurité

La sécurité est un aspect crucial de toute application web, et Symfony fournit un système de sécurité complet et robuste, qui prend en charge plusieurs mécanismes comme l'authentification, la gestion des utilisateurs, le contrôle d'accès et la protection contre les attaques courantes

Un formulaire de connexion basique peut être géré à travers le **LoginAuthenticator** ou d'autres méthodes fournies par Symfony.

**Configuration du formulaire de connexion dans security.yaml** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Le Controller de Connexion** : Le Controller gère les requêtes liées à l'authentification des utilisateurs.

Le login

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Le logout

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**AuthenticationUtils** permet de récupérer les erreurs d'authentification et de pré-remplir le champ du dernier nom d'utilisateur utilisé.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Gestion des Utilisateurs**

Symfony permet une gestion sécurisée des utilisateurs avec des rôles et des permissions définis dans une base de données. La gestion des utilisateurs inclut le stockage des mots de passe hachés et l’attribution de rôles pour contrôler l’accès aux différentes parties de l’application.

1. **Entité User avec Hachage de Mot de Passe** : L'entité **User** peut être configurée pour gérer les informations d'identification (email, mot de passe, rôle, etc.).

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Contrôle d'Accès (ACL)**

Symfony permet de définir des **rôles** pour chaque utilisateur et de restreindre l'accès à certaines routes ou fonctionnalités en fonction de ces rôles.

1. **Définition des rôles dans security.yaml** :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**ROLE\_ADMIN** a accès à toutes les fonctionnalités de **ROLE\_USER**.

Les chemins commençant par **/admin** sont accessibles uniquement aux utilisateurs ayant le rôle **ROLE\_ADMIN**.

Déploiement

**1. Préparation du VPS Hostinger**

1. **Connexion SSH** : J'ai commencé par me connecter à mon **VPS Hostinger** via SSH pour accéder à l'environnement serveur et y exécuter des commandes.



**Installation des dépendances** : J'ai installé les outils nécessaires à l'exécution de mon application Symfony, notamment **PHP**, **MySQL**, **Composer**, ainsi que les modules PHP requis par Symfony.

Une image contenant Police, capture d’écran, texte, ligne

Description générée automatiquement

1. **Configuration de MySQL sur le VPS**
2. **Création de la base de données** : Après l'installation de MySQL, j'ai configuré une base de données locale pour Symfony. Cela implique la création d'une base de données **MySQL**, d'un utilisateur, et l'attribution de privilèges à cet utilisateur.



Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**3. Configuration de MongoDB Atlas**

1. **Connexion à MongoDB Atlas** : J'ai ensuite configuré une base de données **MongoDB** sur MongoDB Atlas pour gérer les données NoSQL. MongoDB Atlas fournit une URL de connexion au format suivant :

***mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retryWrites=true&w=majority***

J'ai récupéré cette **URI** pour la configurer dans mon application Symfony.

Ensuite, j'ai ajouté l'URI de connexion MongoDB dans le fichier .env de mon projet :

***MONGODB\_URL="mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retryWrites=true&w=majority"***

Enfin, j'ai configuré Doctrine MongoDB dans le fichier config/packages/doctrine\_mongodb.yaml pour activer la connexion :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**4. Déploiement de l'application Symfony sur le VPS**

1. **Upload du code source** : J'ai utilisé **Git** pour déployer mon projet Symfony sur le VPS. J'ai cloné mon dépôt Git directement sur le serveur :



1. **Installation des dépendances** : Après avoir cloné le projet, j'ai installé toutes les dépendances nécessaires à Symfony en exécutant la commande **Composer** :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

1. **Configuration du serveur web** : Le serveur utilisé était **Apache**. J'ai configuré un hôte virtuel pour pointer vers le répertoire **public/** de mon application Symfony :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Ensuite, j'ai activé le site et redémarré Apache :

Une image contenant Police, texte, capture d’écran, typographie

Description générée automatiquement

1. **Permissions des fichiers** : J'ai défini les bonnes permissions sur les répertoires de cache et de logs de Symfony pour que le serveur web puisse écrire dans ces répertoires.

Une image contenant capture d’écran, Police, ligne

Description générée automatiquement

**5. Configuration des bases de données**

1. **Configurer la base de données MySQL** : Dans le fichier .env de Symfony, j'ai défini l'URL de connexion MySQL locale pour permettre à Symfony d'utiliser la base de données MySQL sur le VPS :



1. **Configurer MongoDB** : Toujours dans le fichier .env, j'ai ajouté l'URL de connexion MongoDB pour que Symfony puisse se connecter à **MongoDB Atlas** :



1. **Mettre à jour les bases de données** : J'ai ensuite exécuté les migrations pour MySQL afin de créer les tables nécessaires dans la base de données MySQL :



**6. Finalisation du déploiement**

1. **Nettoyage du cache et optimisation** : Pour optimiser Symfony en production, j'ai nettoyé et optimisé le cache avec la commande suivante :



1. **Test de l'application** : Après le déploiement, j'ai accédé à l'application via l'URL associée au serveur pour m'assurer que tout fonctionnait correctement, notamment les interactions avec les bases de données MySQL et MongoDB Atlas.

**Conclusion**

Le déploiement de l'application Symfony sur un VPS Hostinger avec **MySQL** local et **MongoDB Atlas** a impliqué les étapes suivantes :

* Configuration du VPS pour PHP, MySQL et Composer.
* Connexion et configuration de MongoDB Atlas via Doctrine MongoDB ODM.
* Déploiement du code source Symfony sur le serveur via Git.
* Configuration du serveur web Apache pour pointer vers le répertoire public de Symfony.
* Configuration des bases de données MySQL et MongoDB, ainsi que des migrations et optimisations.

Ce processus a permis de déployer une application Symfony fonctionnelle, connectée à une base de données relationnelle MySQL et une base NoSQL MongoDB.

Docker

**Docker** est une plateforme open-source qui permet de créer, déployer, et exécuter des applications dans des conteneurs. Docker simplifie la gestion des environnements en permettant de développer, tester et déployer des applications dans des environnements isolés, reproductibles et cohérents.

**Pourquoi Docker ?**

L’utilisation de Docker dans le projet **Arcadia** apporte plusieurs avantages :

* **Portabilité** : Grâce aux conteneurs, l’application peut être déployée et exécutée sur différentes machines avec la même configuration, sans se soucier des différences entre environnements.
* **Isolation des environnements** : Chaque conteneur fonctionne indépendamment, ce qui permet de séparer les différents services (comme la base de données, le serveur web, etc.) et de garantir une meilleure stabilité et sécurité.
* **Simplification du déploiement** : Docker permet de créer des images d’applications qui peuvent être déployées facilement dans différents environnements, de la machine de développement à la production.

**Installation de Docker**

J’ai téléchargé Docker pour **Windows** à partir du site officiel :  
<https://docs.docker.com/desktop/install/windows-install/>

Après l’installation, j’ai procédé à la configuration de Docker pour mon projet. Cela inclut la création de fichiers Docker spécifiques pour définir les services nécessaires à l’exécution de l’application **Arcadia**.

**Configuration de Docker : Fichier docker-compose.yml**

J'ai configuré Docker pour gérer plusieurs services au sein du projet via le fichier **docker-compose.yml**. Ce fichier permet de définir et gérer les applications **multi-conteneurs**. Voici ce qu'il permet :

* **Définition des services** : Le fichier docker-compose.yml décrit les différents services nécessaires à l’application, tels que la base de données (MySQL), le serveur web (Nginx), et l'application Symfony. Chaque service est défini avec ses paramètres spécifiques (image Docker, ports, volumes, variables d’environnement, etc.).
* **Coordination des conteneurs** : Grâce à ce fichier, Docker orchestre l’exécution de plusieurs conteneurs. Chaque service est lancé de manière indépendante, mais ils sont configurés pour communiquer entre eux.
* **Simplification des commandes** : En utilisant Docker Compose, une seule commande est nécessaire pour démarrer tous les conteneurs :



Cette commande simplifie le processus de déploiement et de gestion de l’environnement, en évitant d’avoir à démarrer manuellement chaque conteneur séparément.

Voici un extrait du fichier **docker-compose.yml** utilisé dans le projet **Arcadia** :

Une image contenant texte, Police, capture d’écran, conception

Description générée automatiquementUne image contenant texte, capture d’écran, Police, conception

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Dockerfile**

En plus du fichier **docker-compose.yml**, le projet **Arcadia** utilise un **Dockerfile** pour définir l'image Docker de l’application. Le **Dockerfile** est un fichier texte contenant toutes les instructions nécessaires pour assembler une image Docker spécifique à l’application, en spécifiant les dépendances, les configurations, et les scripts nécessaires pour faire fonctionner l'application dans un conteneur.

**Fonctionnement du Dockerfile**

Le **Dockerfile** contient les étapes suivantes :

1. **Image de base** : L'image de base utilisée est généralement une version allégée d'un système d'exploitation ou d'une plateforme spécifique. Dans mon cas, j'ai choisi une image PHP avec FPM (FastCGI Process Manager) pour exécuter l'application **Symfony**.

Une image contenant texte, Police, capture d’écran, nombre

Description générée automatiquement

1. **Installation des dépendances** : Le Dockerfile inclut les instructions pour installer les dépendances nécessaires à l’application, comme des bibliothèques PHP spécifiques ou des outils de gestion comme Composer.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

1. **Ajout des fichiers** : Les fichiers du projet (par exemple, le code Symfony) sont copiés à l’intérieur de l’image Docker pour permettre leur exécution.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. **Configuration de l’environnement** : Le Dockerfile permet également de configurer l’environnement à l’intérieur du conteneur, par exemple, en définissant le répertoire de travail ou les droits d'accès.

Une image contenant texte, Police, capture d’écran, typographie

Description générée automatiquement

1. **Exécution de commandes** : Le Dockerfile peut exécuter des commandes comme l’installation des dépendances avec Composer ou la compilation d'actifs front-end.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Le fichier « default.conf » qui est un fichier de configuration pour configurer Nginx.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Le fichier « default.conf » qui est un fichier de configuration pour configurer Nginx.

Une fois la commande « docker-compose up » lancée, je vérifie sur docker que mes containers soit bien créé.

Une image contenant capture d’écran, texte

Description générée automatiquement

**Conclusion**

Le **Dockerfile** est un élément clé dans la gestion de l'environnement de développement et de production du projet **Arcadia**. Il permet de créer une image Docker qui contient toutes les dépendances nécessaires à l’exécution de l’application Symfony, garantissant que l’environnement reste stable et reproductible, quel que soit l'endroit où l'application est déployée.

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, cercle

Description générée automatiquement

Une image contenant texte, diagramme, capture d’écran, ligne

Description générée automatiquement

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, diagramme

Description générée automatiquement

Une image contenant texte, capture d’écran, Site web, Page web

Description générée automatiquement