



Dossier **Projet**

Titre RNCP – Développeur Web / Web Mobile

Leliard René

Studi – Promotion Novembre/Décembre 2024

Table des matières

3 =>	A propos de moi
4 =>	Compétences du référentiel couverte par le projet
4 =>	Présentation du projet
5 =>	Cahier des charges
7 =>	Contraintes
8 =>	Technologies, outils, Framework
10 =>	Gestion de projet
12 =>	Conception
17 =>	Charte graphique
18 =>	Wireframe et maquettes
19 =>	Installation et configuration de l'environnement de travail
24 =>	Développement
43 =>	Mailling
47 =>	Sécurité
51 =>	Déploiement
56 =>	Intégration et explication du CI/CD
58 =>	Docker
65 =>	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 reconverter 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

2. 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 déployé.

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

- **Contrôle d'accès:** 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



Trello pour la gestion de projet et de tâches



draw.io

Draw.io pour réaliser mon diagramme de cas d'utilisation et de séquence




dbdiagram.io

DBdiagram.io pour réaliser mon diagramme MLD




Figma pour créer mes wireframes et maquettes


2. Technologie Frontend




HTML5 pour structurer et présenter le contenu de mon application sur le web



Bootstrap pour créer mes interfaces utilisateur rapidement



Javascript pour rendre mes pages web dynamiques



Moteur de template intégré au framework Symfony, avec une syntaxe claire et sécurisée par défaut.

3. Backend



Symfony utilisé comme Framework pour le développement avec PHP



PHP pour développer l'application côté serveur



Mailjet utilisé pour la gestion des e-mails



Docker utilisé comme outil de conteneurisation

4. Bases de données



MongoDB utilisé pour mesurer la popularité des animaux. Ces données seront accessibles en lecture/écriture très régulièrement



MySQL utilisé pour ma base de données relationnelle



PhpMyAdmin utilisé comme interface pour gérer ma base de données relationnelle



SQL utilisé pour exploiter ma base de données relationnelle

5. Développement, tests et déploiement



PHPStorm un IDE idéal pour coder en PHP



Hostinger utilisé pour déployer mon application



Atlas MongoDB pour gérer la base NoSQL dans le cloud

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.

2. 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 68, 69)

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.

Annexe page 70/71

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.

2. 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.

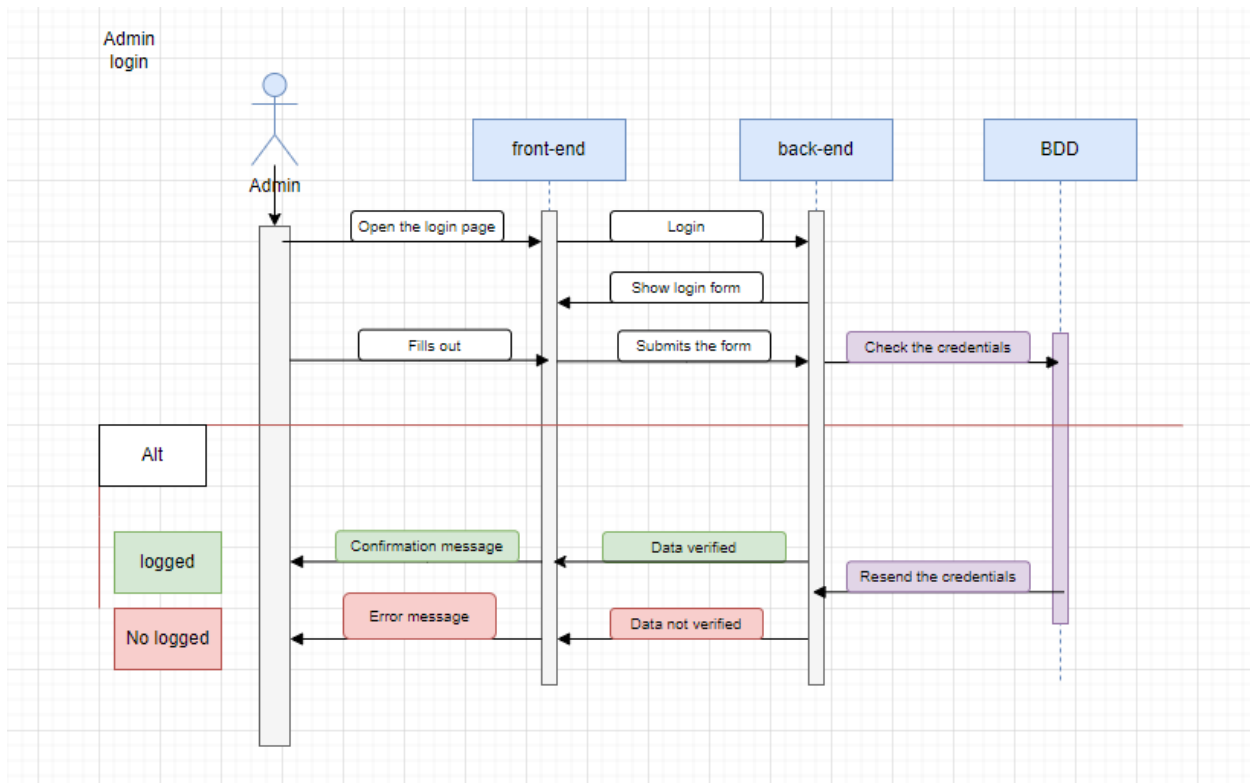
3. Résultats de la vérification

- **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.



3. 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

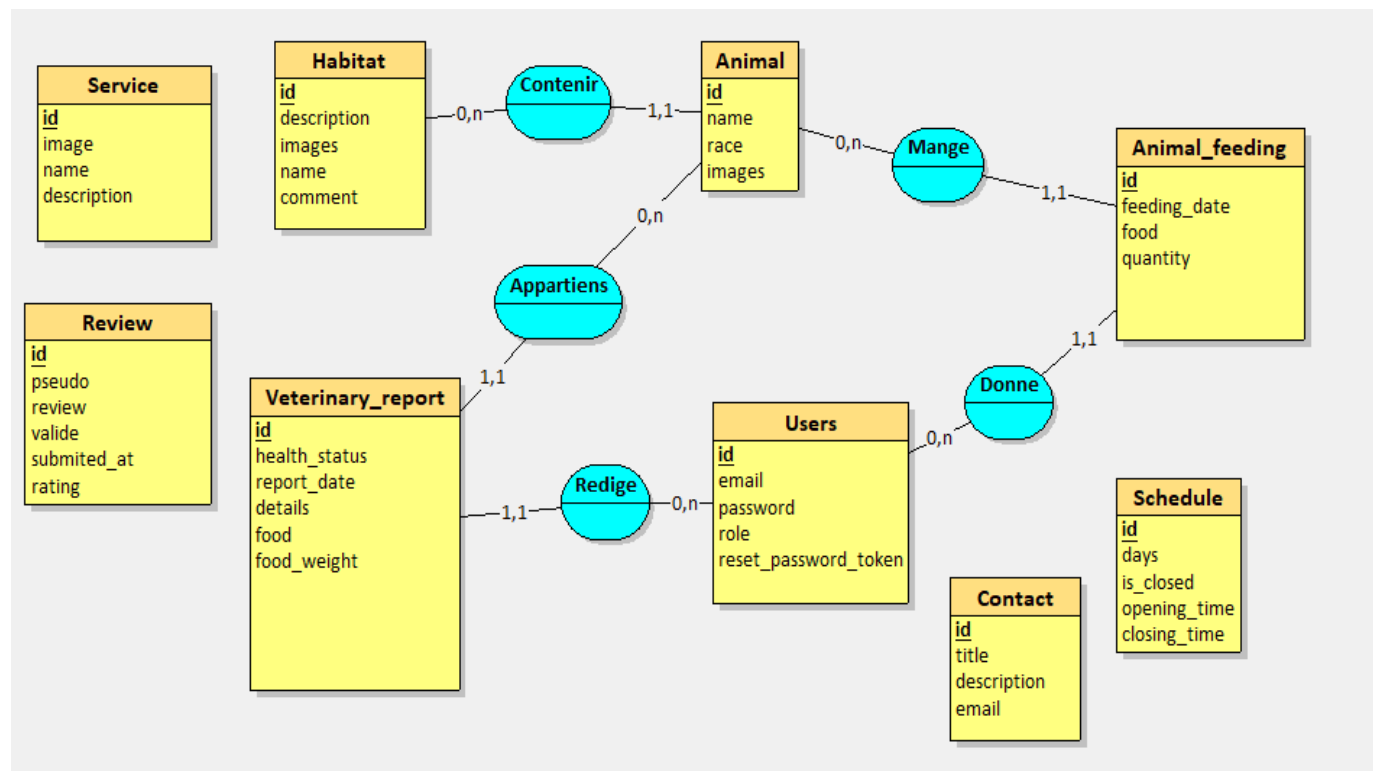
- **Service** : Représente les différents services proposés par le parc (boutique, restauration, etc.). Chaque service est identifié par un id unique et possède un nom, une image et une description.
- **Habitat** : Représente les habitats des animaux. Chaque habitat est décrit par son id, un nom, une 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.



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.



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 73)

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 74)

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**.

2. 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** :

```
PS C:\wamp64> cd www  
PS C:\wamp64\www> symfony new arcadia --webapp
```

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 :

```
PS C:\wamp64\www> cd studi-ecf-arcadia  
PS C:\wamp64\www\studi-ecf-arcadia> symfony serve
```

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.

3. 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 réalisé, 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

4. 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.

5. 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 créé un compte puis un projet et j'ai copié les informations de connexion à ma base de données dans mon projet.

6. 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 :



```

1 <!DOCTYPE html>
2 <html lang="fr-FR">
3   <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1">
6     <title>{% block title %}Ecf Arcadia!{% endblock %}</title>

```

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


```

1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <div class="row mb-4 m-4">
5          <div class="col-lg-8">
6              <div class="card mb-3 rounded-4 cadre">
7                  <div class="row g-0">
8                      {# Section présentation #}

```

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.

Annexe pages 67

3. 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.

4. 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:

```
1 document.addEventListener( type: 'DOMContentLoaded', listener: () : void => {
```

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 :

```
document.querySelectorAll( selectors: '.habitat-link').forEach( callbackfn: link : Element => {
```

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 :

```
link.addEventListener( type: 'click', listener: (event : Event) : void => {
  event.preventDefault();
```

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 :

```
const habitatId : string = link.getAttribute( qualifiedName: 'data-habitat-id');
```

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 :

```
fetch( input: `/habitat/${habitatId}/animals`) Promise<Response>
```

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 :

```
.then(response : Response => {
  //console.log('Response:', response);
  if (!response.ok) {
    throw new Error(`HTTP error! Status: ${response.status}`);
  }
  return response.json();
}) Promise<any>
```

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 :

```
.then(data => {
  //console.log('Data:', data);
  const animalsList : HTMLElement = document.getElementById( elementId: 'animals-list');
  animalsList.innerHTML = '';
})
```

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 :

```
const habitatDescriptionDiv : HTMLDivElement = document.createElement( tagName: 'div');
habitatDescriptionDiv.className = 'col-lg-12';
habitatDescriptionDiv.innerHTML = `
  <div class="card m-4 cadre">
    <div class="card-body">
      <h3 class="card-title">Description de l'habitat (${data.habitat_name})</h3>
      <p>${data.habitat_detail}</p>
    </div>
  </div>
`;
animalsList.appendChild(habitatDescriptionDiv);
```

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 :

```
data.animals.forEach( animal => {
  const animalDiv : HTMLDivElement = document.createElement( tagName: 'div');
  animalDiv.className = 'col-lg-4';
  const lastReport : any | null = animal.veterinary_reports.length > 0 ? animal.veterinary_reports[animal.veterinary_reports.length - 1] : null;
  animalDiv.innerHTML = `
    <div class="card m-4 cadre detail">
      <div class="card-body">
        <h3 class="card-title">${animal.name}</h3>
        
        <h4>Rapport Vétérinaire</h4>
        ${lastReport ? `
          <div>
            <p><strong>État de santé:</strong> ${lastReport.health_status}</p>
            <p><strong>Nourriture:</strong> ${lastReport.food}</p>
            <p><strong>Poids de la nourriture:</strong> ${lastReport.food_weight} kg</p>
            <p><strong>Date du rapport:</strong> ${lastReport.report_date}</p>
            <p><strong>Détail:</strong> ${lastReport.detail}</p>
          </div>
        ` : '<p>Aucun rapport vétérinaire disponible.</p>'}
      </div>
    </div>
  `;
  animalsList.appendChild(animalDiv);
});
```

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 :

```
animalsList.scrollToView( arg: {behavior: 'smooth'});
}) Promise<void>
```

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 :

```
.catch(error => {
    console.error('Error fetching habitat details:', error.message);
});
```

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.

5. 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

```
#ORM\Entity(repositoryClass: AnimalRepository::class)]
class Animal MrLeoufff, 05/07/2024 11:31 • First commit
{
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    2 usages
    #[ORM\Column(length: 150, nullable: false)]
    private ?string $name = null;

    2 usages
    #[ORM\Column(length: 150, nullable: false)]
    private ?string $race = null;
```

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.

```
#[ORM\ManyToOne(targetEntity: Habitat::class, inversedBy: 'animals')]
#[ORM\JoinColumn(nullable: false)]
private Habitat $habitat;
```

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 remplacé par l'identifiant (\$habitatId) lors de l'exécution.

Script création BDD pages 75/76/77

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.

6. 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

```
class AnimalController extends AbstractController
{
  ⚡ MrLeoufff
  #[Route('/veto/animal/', name: 'app_animal_index', methods: ['GET'])]
  public function index(AnimalRepository $animalRepository): Response
  {
    return $this->render( view: 'animal/index', [
      'animals' => $animalRepository->findAll(),
    ]);
  }
}
```

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

```
#[Route('/veto/animal/new', name: 'app_animal_new', methods: ['GET', 'POST'])]
public function new(
    Request $request,
    EntityManagerInterface $entityManager,
    HabitatRepository $habitatRepository,
    SluggerInterface $slugger
): Response {
    $habitat = $habitatRepository->findOneBy([]);
    if (!$habitat) {
        throw $this->createNotFoundException('Aucun habitat trouvé, veuillez en créer un avant de créer');
    }
}
```

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 pages(65/66)

Route /animal - Liste des Animaux

```
#[Route('/animal', name: 'app_animal_list')]
public function list(
    EntityManagerInterface $entityManager,
): Response
{
    $animals = $entityManager->getRepository(Animal::class)->findAll();

    return $this->render( view: 'animal/list', [
        'animals' => $animals,
    ]);
}
```

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

```
#[Route('/veto/animal/{id}/edit', name: 'app_animal_edit', methods: ['GET', 'POST'])]
public function edit(
    Request $request,
    Animal $animal,
    EntityManagerInterface $entityManager,
    SluggerInterface $slugger,
    HabitatRepository $habitatRepository
): Response
{
    $form = $this->createForm( type: AnimalType::class, $animal);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $images = $form->get('image')->getData();
        $imageNames = [];
    }
}
```

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

```
#[Route('/veto/animal/{id}/', name: 'app_animal_delete', methods: ['POST'])]
public function delete(
    Request $request,
    Animal $animal,
    EntityManagerInterface $entityManager
): Response
{
    if ($this->isCsrfTokenValid( id: 'delete' . $animal->getId(), $request->request->get( key: '_token' ))) {
        $entityManager->remove($animal);
        $entityManager->flush();

        $this->addFlash( type: 'success', message: 'Animal supprimé avec succès.' );
    } else {
        $this->addFlash( type: 'error', message: 'Échec de la suppression de l\'animal.' );
    }

    return $this->redirectToRoute( route: '/veto/animal/' );
}
```

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

```
#[Route('/animal/{id}', name: 'app_animal_show', methods: ['GET'])]
public function show(
    Animal $animal,
    DocumentManager $documentManager,
    VeterinaryReportRepository $veterinaryReportRepository,
    AnimalFeedingRepository $animalFeedingRepository
): Response
{
    $animalView = $documentManager->getRepository(class: AnimalView::class)->findOneBy(['animalName' => $animal->getName()]);

    if (!$animalView) {
        $animalView = new AnimalView();
        $animalView->setAnimalName($animal->getName());
        $documentManager->persist($animalView);
    }
}
```

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 p78

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.

7. 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

```
class MailerService
{
    3 usages
    private string $zooEmail;
    3 usages
    private MailerInterface $mailer;
```

\$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

```

public function __construct(
    #[Autowire('%zoo_email%')]string $zooEmail,
    MailerInterface $mailer
) {
    $this->zooEmail = $zooEmail;
    $this->mailer = $mailer;
}

```

#[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()

```

public function sendContactEmail(Contact $contact):void | MrLeoufff, 13/07/2024 19
{
    $email = (new NotificationEmail())
        ->from($contact->getEmail())
        ->to($this->zooEmail)
        ->subject( subject: 'Nouvelle demande de contact: ' . $contact->getTitle())
        ->htmlTemplate( template: 'emails/contact')
        ->context([
            'title' => $contact->getTitle(),
            'description' => $contact->getDescription(),
            'sender_email' => $contact->getEmail(),
        ]);

    $this->mailer->send($email);
}

```

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()

```
public function sendUserCreationEmail(User $user): void
{
    $email = (new NotificationEmail())
        ->from($this->zooEmail)
        ->to($user->getEmail())
        ->subject('Votre compte a été créé')
        ->htmlTemplate('emails/user_creation')
        ->context([
            'user_email' => $user->getEmail(),
        ]);

    $this->mailer->send($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. □ **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` :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
<p>Titre: {{ title }}</p>
<p>Description: {{ description }}</p>
<p>Email de l'expéditeur: {{ sender_email }}</p>
</body>
</html>
```

Exemple du fichier `user_creation.html.twig` :

```
<!DOCTYPE html> MrLeoufff, 13/07/2024 19:36 • add mailing
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
<p>Bonjour,</p>
<p>Votre compte a été créé avec succès. Votre nom d'utilisateur est {{ user_email }}.</p>
<p>Veuillez contacter l'administrateur pour obtenir votre mot de passe.</p>
</body>
</html>
```

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 :

```

security:
  # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
  password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
  # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
  providers:
    app_user_provider:
      entity:
        class: App\Entity\User
        property: email
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      lazy: true
      provider: app_user_provider
      form_login:
        login_path: app_login
        check_path: app_login
      logout:
        path: app_logout
        target: /

```

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

Le login

```

#[Route('/login', name: 'app_login')]
public function login(AuthenticationUtils $utils): Response
{
    if ($this->getUser()) {
        return $this->redirectToRoute('route: /');
    }

    $error = $utils->getLastAuthenticationError();
    $lastUsername = $utils->getLastUsername();

    if ($error) {
        $this->addFlash('error', 'Échec de la connexion. Veuillez vérifier vos identifiants.');
```

Le logout

```
#[Route('/logout', name: 'app_logout')]
public function logout(): void
{
    throw new \LogicException( message: '');
```

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

```
{% block body %}
<h1>Connexion</h1>
<form action="{{ path('/login') }}" method="post">
    <label for="username">Email:</label>
    <input class="input-group-text" type="text" id="username" name="_username" value="{{ last_username }}" required autofocus>

    <label for="password">Mot de passe:</label>
    <input class="input-group-text" type="password" id="password" name="_password" required>

    <button class="btn btn-primary" type="submit">Login</button>

    {% if error %}
        <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}
</form>
{% endblock %}
```

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.).

```
#[ORM\Entity(repositoryClass: UserRepository::class)]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
```

```

#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;

3 usages
#[ORM\Column(length: 255)]
private ?string $email = null;

2 usages
#[ORM\Column(length: 255)]
private ?string $password = null;

2 usages
#[ORM\Column(type: 'string', length: 64, nullable: true)]
private ?string $resetPasswordToken;

2 usages
private ?string $plainPassword = null;

2 usages
#[ORM\Column]
private array $roles = [];

```

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 :

```
access_control:| MrLeoufff, 04/07/2024 14:27 • Add webapp packa
  - { path: ^/admin, roles: ROLE_ADMIN }

  - { path: ^/employee, roles: ROLE_EMPLOYEE}
  #- { path: ^/employee, roles: ROLE_ADMIN }

  - { path: ^/veto, roles: ROLE_VETERINARIAN }
  #- { path: ^/veto, roles: ROLE_ADMIN }
  - { path: ^/public, roles: IS_AUTHENTICATED_ANONYMOUSLY }

role_hierarchy:
  ROLE_ADMIN: [ ROLE_VETERINARIAN, ROLE_EMPLOYEE ]
```

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.

```
ssh user@your_vps_ip
```

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.

```
sudo apt update
sudo apt install php libapache2-mod-php php-mysql php-xml
```

2. Configuration de MySQL sur le VPS

1. **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.

```
sudo mysql -u root -p
```

```
CREATE DATABASE symfony_db;
CREATE USER 'symfony_user'@'localhost' IDENTIFIED BY 'your_password';
GRANT ALL PRIVILEGES ON symfony_db.* TO 'symfony_user'@'localhost';
FLUSH PRIVILEGES;
```

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 :

```
doctrine_mongodb:
  connections:
    default:
      server: '%env(MONGODB_URL)%'
      options: {}
  default_database: <dbname>
  document_managers:
    default:
      auto_mapping: true
```

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 :

```
git clone https://github.com/username/your-symfony-project.git /var/www/symfony
```

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

```
cd /var/www/symfony
composer install --no-dev --optimize-autoloader
```

3. **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 :

```
<VirtualHost *:80>
    ServerName yourdomain.com
    DocumentRoot /var/www/symfony/public

    <Directory /var/www/symfony/public>
        AllowOverride All
        Order Allow,Deny
        Allow from All
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/symfony_error.log
    CustomLog ${APACHE_LOG_DIR}/symfony_access.log combined
</VirtualHost>
```

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

```
sudo a2ensite symfony
sudo systemctl reload apache2
```

4. **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.

```
sudo chown -R www-data:www-data /var/www/symfony
sudo chmod -R 775 /var/www/symfony/var/cache /var/www/symfony/var/log
```

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 :

```
DATABASE_URL="mysql://symfony_user:your_password@127.0.0.1:3306/symfony_db?serverVe
```

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

```
MONGODB_URL="mongodb+srv://<username>:<password>@cluster0.mongodb.net/<dbname>?retr
```

3. **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 :

```
php bin/console doctrine:migrations:migrate
```

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 :

```
php bin/console cache:clear --env=prod --no-debug
```

2. **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.

Résilience et gestion des sauvegardes

Qu'est-ce que la Résilience ?

La **résilience informatique** est la capacité d'un système à continuer de fonctionner, même en cas de pannes ou de perturbations. Pour un site web ou une application, cela signifie garantir la continuité du service, la sécurité des données et une récupération rapide en cas de problème.

Chez Hostinger, la résilience est au cœur des services VPS, permettant aux développeurs et entreprises de s'appuyer sur une infrastructure fiable et performante.

Sauvegarde et Compression Automatiques de la Base de Données

Lorsque vous choisissez un **VPS (Virtual Private Server)** chez Hostinger, vous bénéficiez de fonctionnalités avancées pour gérer vos bases de données de manière sécurisée et efficace :

1. Sauvegarde Automatique

Hostinger propose des sauvegardes automatiques régulières pour assurer la continuité de vos services :

- **Fréquence configurable** : Les sauvegardes peuvent être planifiées quotidiennement, hebdomadairement ou selon vos besoins spécifiques.
- **Restauration en un clic** : En cas de problème, vous pouvez restaurer votre base de données depuis l'interface utilisateur Hostinger, réduisant les temps d'arrêt au minimum.

- **Emplacement sécurisé** : Les sauvegardes sont stockées sur des serveurs externes, garantissant la sécurité même en cas de défaillance matérielle.

2. Compression des Données

Pour optimiser l'espace de stockage et accélérer les processus de sauvegarde :

- Les bases de données sont **compressées automatiquement** au format gzip ou zip.
- Cette compression permet de réduire considérablement l'espace nécessaire pour les sauvegardes, tout en facilitant leur transfert ou téléchargement.

Avantages de ces Services

1. Sécurité renforcée :

- Vos données sont protégées contre les pertes accidentelles ou les cyberattaques.
- Les sauvegardes externes empêchent les corruptions dues à des pannes matérielles.

2. Économie de ressources :

- La compression réduit la taille des sauvegardes, optimisant l'espace disque et minimisant les coûts de stockage.

3. Fiabilité pour les développeurs :

- En cas de défaillance ou d'erreur humaine, vous pouvez facilement restaurer vos données et éviter des pertes de revenus.

Résilience grâce à Hostinger et Symfony

En combinant les capacités de sauvegarde de Hostinger et la robustesse de Symfony, votre application peut atteindre un haut niveau de résilience :

- Symfony offre une gestion avancée des erreurs et des logs, permettant de détecter rapidement les incidents.
- Hostinger garantit la disponibilité et l'intégrité des données, assurant une reprise rapide après un incident.

Intégration et explication du CI/CD

Dans cette section, je décris les pipelines CI/CD utilisés dans le projet, détaillant les étapes de déploiement continu (Continuous Deployment) et de tests continus (Continuous Testing) configurés avec GitHub Actions.

1. Pipeline de Tests (CI)

Ce pipeline garantit que les modifications apportées à la branche dev respectent les standards de qualité avant leur fusion. Il exécute des tests Symfony via PHPUnit dans un environnement isolé.

Workflow :

- **Événement déclencheur** : Le pipeline se déclenche sur :
 - Un push vers la branche dev.
 - Une Pull Request vers dev.
- **Étapes principales** :
 - **Configuration PHP** : Installation de PHP 8.3 avec l'action shivammathur/setup-php.
 - **Installation des dépendances** : Utilisation de Composer pour installer les bibliothèques nécessaires.
 - **Configuration de la base de données** : Création d'une base SQLite pour les tests.
 - **Nettoyage et configuration** : Effacement du cache Symfony et configuration des variables d'environnement pour les tests (.env.test.local).
 - **Exécution des tests** : Les tests unitaires et fonctionnels sont exécutés avec PHPUnit.

Annexe en page 79

2. Pipeline de Déploiement (CD)

Ce pipeline est déclenché automatiquement lors d'un push sur la branche main. Il s'assure que l'application est déployée sur un environnement de production via SSH.

Workflow :

- **Événement déclencheur** : Le pipeline se déclenche sur un push vers la branche principale main.
- **Configuration de l'environnement** : Variables sensibles (comme SSHPASS, SSH_USER, et SSH_IP) sont gérées via les secrets de GitHub.
- **Étapes principales** :
 - **Récupération du code** : Le dépôt est cloné grâce à l'action checkout.
 - **Installation de sshpass** : Cet outil est nécessaire pour exécuter des commandes SSH avec un mot de passe sécurisé.
 - **Connexion et déploiement** : La commande sshpass permet de se connecter au serveur distant pour :
 - Accéder au répertoire du projet (htdocs/reneleliard.online).
 - Mettre à jour le code avec git pull.
 - Installer les dépendances avec composer install.

Annexe en page 80

3. Résumé des Avantages

- **Automatisation** : Les pipelines CI/CD automatisent le déploiement et les tests, réduisant les erreurs humaines.
- **Sécurité** : Les secrets de GitHub protègent les informations sensibles.
- **Qualité** : Les tests automatiques assurent la stabilité du code avant son intégration en production.
- **Efficacité** : Le pipeline permet un déploiement rapide et fiable avec des feedbacks immédiats sur les erreurs.

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 :

```
PS C:\wamp64\www\studi-ecf-arcadia> docker-compose up
```

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** :

```
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./public:/var/www/html
    depends_on:
      - php
    networks:
      - arcadia-net
```

```
php:
  image: php:8.3-fpm
  volumes:
    - ./app:/var/www/html
  networks:
    - arcadia-net
```

```
db:
  image: mysql:8.0
  environment:
    MYSQL_ROOT_PASSWORD: secret
    MYSQL_DATABASE: arcadia
    MYSQL_USER: user
    MYSQL_PASSWORD: password
  volumes:
    - db_data:/var/lib/mysql
  networks:
    - arcadia-net
```

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 :

7. **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**.

```
FROM php:8.3.9-fpm
LABEL authors="René"
```

8. **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.

```
# Installer Composer
COPY --from=composer:2.7.7 /usr/bin/composer /usr/bin/composer
```

```
# Installer les extensions PHP nécessaires
RUN apt update && apt install -y \
    libpng-dev \
    libjpeg-dev \
    libzip-dev \
    libfreetype6-dev \
    libonig-dev \
    libxslt1-dev \
    unzip \
    build-essential \
    git \
    nginx \
    libssl-dev \
    default-mysql-client \
    && docker-php-ext-configure gd --with-freetype --with-jpeg \
    && docker-php-ext-install -j$(nproc) gd \
    && docker-php-ext-install pdo_mysql \
    && docker-php-ext-enable opcache \
    && docker-php-ext-install xsl \
    && docker-php-ext-install zip \
    && docker-php-ext-install intl \
    && docker-php-ext-install soap \
    && pecl install mongodb \
    && docker-php-ext-enable mongodb
```

9. **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.

```
COPY . .

COPY build/nginx/conf/default.conf /etc/nginx/conf.d/default.conf
COPY build/php/custom.ini /usr/local/etc/php/conf.d/

COPY entrypoint.sh /usr/local/bin/entrypoint.sh
COPY build/supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

10. **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.

```
# Copier les fichiers de l'application
WORKDIR /var/www/html
```

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

```
# Rendre le script exécutable
RUN chmod +x /usr/local/bin/entrypoint.sh
RUN mv "$PHP_INI_DIR/php.ini-production" "$PHP_INI_DIR/php.ini"

# Installer les dépendances Composer
ENV COMPOSER_ALLOW_SUPERUSER=1
RUN composer install --no-interaction --optimize-autoloader
```

```
# Démarrer Apache
# CMD service nginx start && php-fpm
# Utiliser supervisord pour gérer les processus
RUN apt-get install -y supervisor
COPY build/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

ENTRYPOINT ["entrypoint.sh"]
CMD ["supervisord", "-c", "/etc/supervisor/conf.d/supervisord.conf"]
```

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

```

server {
    listen 80;

    server_name localhost;

    root /var/www/html/public;
    index index.php index.html index.htm;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_pass web:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }

    location ~ /\.ht {
        deny all;
    }
}

```

mrleoufff, 19/07/2024 15:44 • test docker

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.

<input type="checkbox"/>	Name	Image	Status	Port(s)	CPU (%)	Last started
<input type="checkbox"/>	arcadia		Running (5/5)		1.15%	1 day ago
<input type="checkbox"/>	mysql-1	mysql:latest	Running		0.59%	1 day ago
<input type="checkbox"/>	phpmyadmin-1	phpmyadmin/phpmyadmin	Running	5001:80	0%	1 day ago
<input type="checkbox"/>	php-1	arcadia-php	Running	5173:5173	0.17%	1 day ago
<input type="checkbox"/>	mongodb-1	mongo	Running		0.39%	1 day ago
<input type="checkbox"/>	app-1	arcadia-app	Running	5000:80	0%	1 day ago

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.


```

#[Route('/veto/animal/new', name: 'app_animal_new', methods: ['GET', 'POST'])] MrLeoufff, 14/07/2024 23:14 • detail
public function new(
    Request $request,
    EntityManagerInterface $entityManager,
    HabitatRepository $habitatRepository,
    SluggerInterface $slugger
): Response
{
    $habitat = $habitatRepository->findOneBy([]);
    if (!$habitat) {
        throw $this->createNotFoundException('Aucun habitat trouvé, veuillez en créer un avant de créer un animal.');
```

```
        }  
        $imageNames [] = $newFilename;  
    }  
    $animal->setImage($imageNames);  
} else {  
    $this->addFlash( type: 'error', message: 'Aucune image trouvée.');
```



```
    $entityManager->persist($animal);  
    $entityManager->flush();  
  
    $this->addFlash( type: 'success', message: 'Animal créé avec succès avec des images.');
```



```
    return $this->redirectToRoute( route: '/animal');
```



```
}  
  
return $this->render( view: 'animal/new', [  
    'form' => $form->createView(),  
]);  
}
```

```

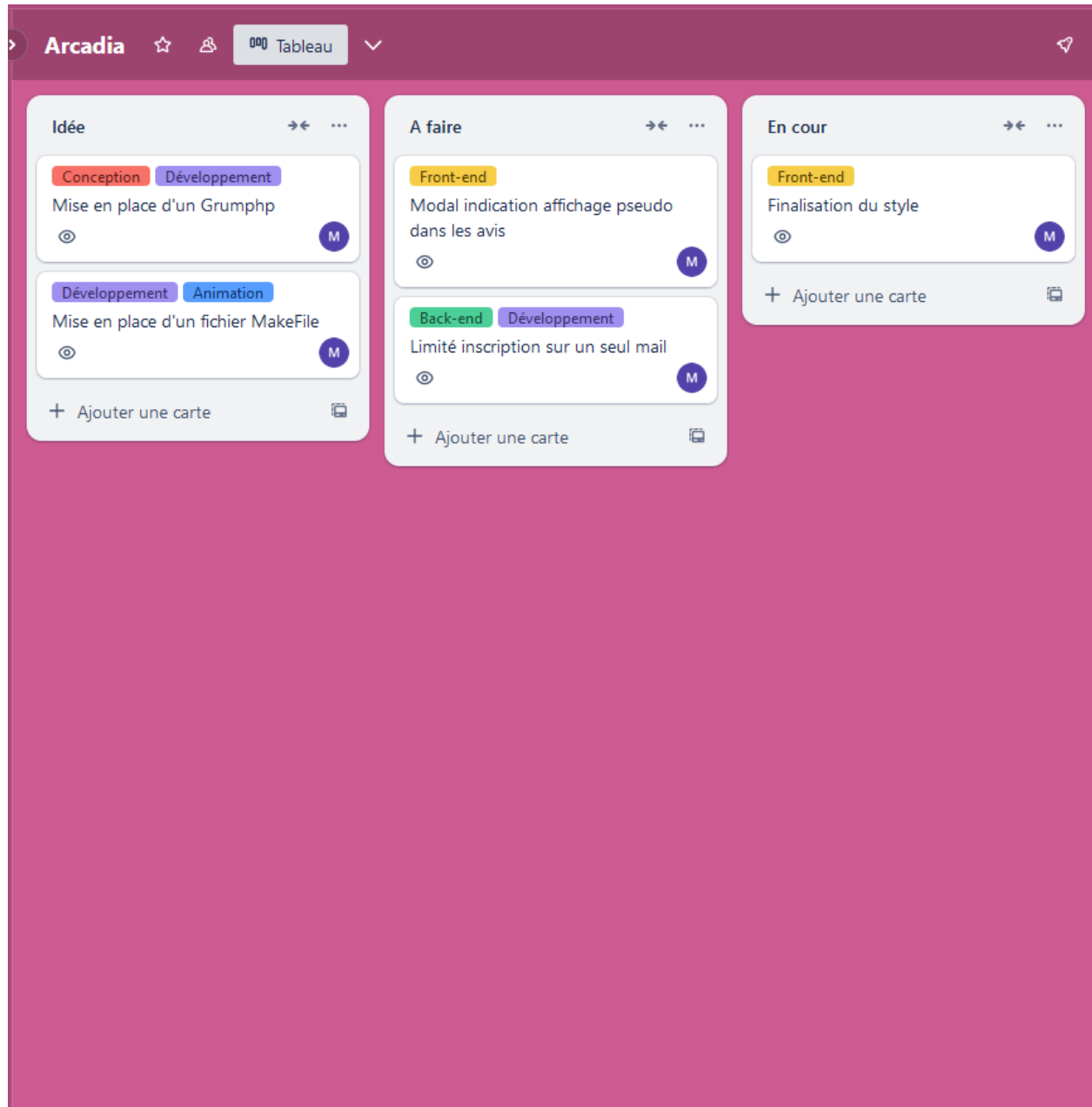
{# Section Horaires #}

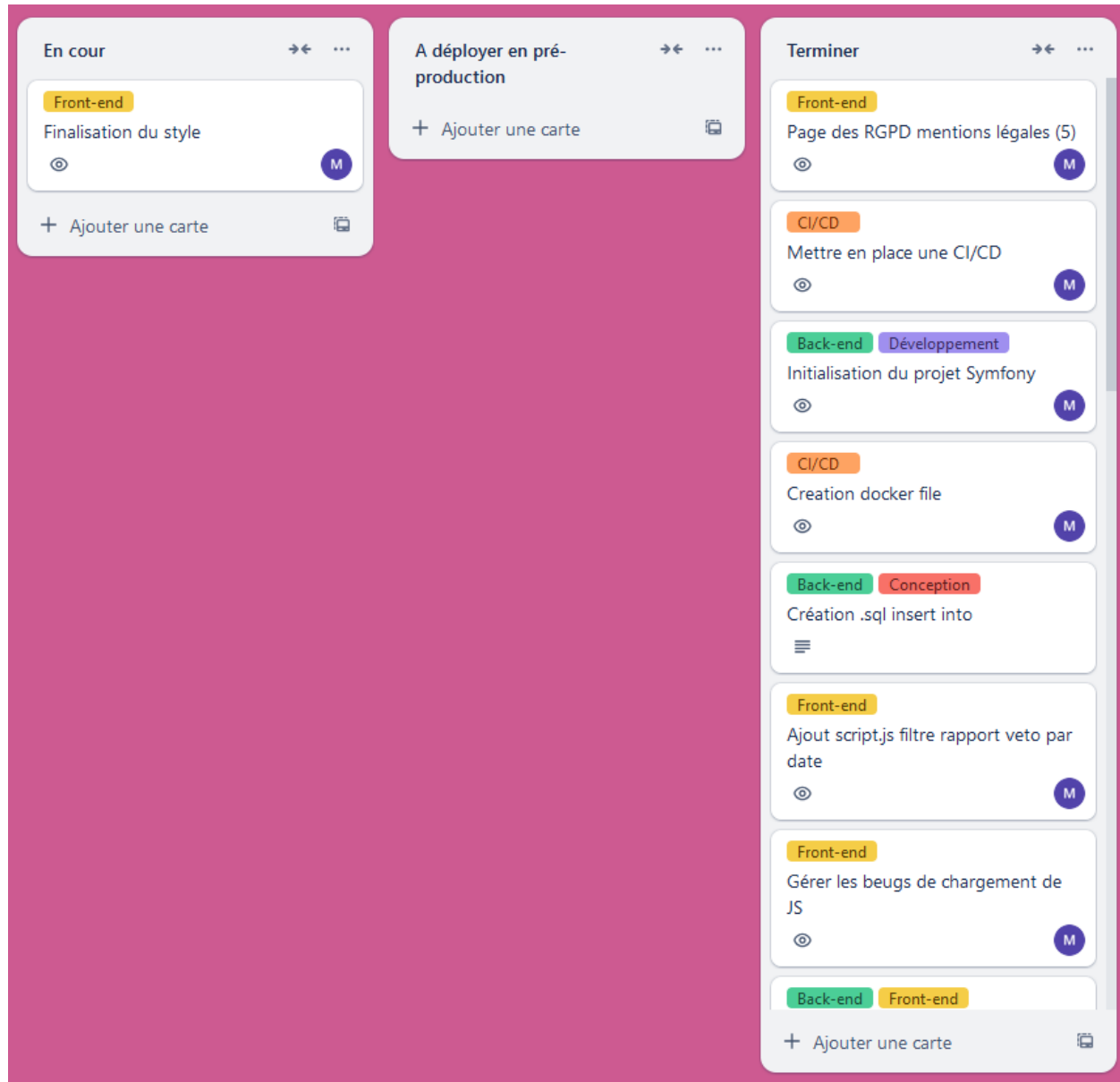
<div class="card container-fluid mt-3 mt-xxl-5">
  <table class="table avis">
    <thead>
      <tr>
        <th>Jours </th>
        <th>Heure d'ouverture</th>
        <th>Heure de fermeture</th>
      </tr>
    </thead>
    <tbody>
      {% for schedule in schedules %}
        <tr>
          <td>{{ schedule.day }}</td>
          {% if schedule.isclosed %}
            <td>Fermé</td>
            <td>Fermé</td>
          {% else %}
            <td>{{ schedule.openingTime|date('H:i') }}</td>
            <td>{{ schedule.closingTime|date('H:i') }}</td>
          {% endif %}
        </tr>
      {% endfor %}
    </tbody>
  </table>
</div>

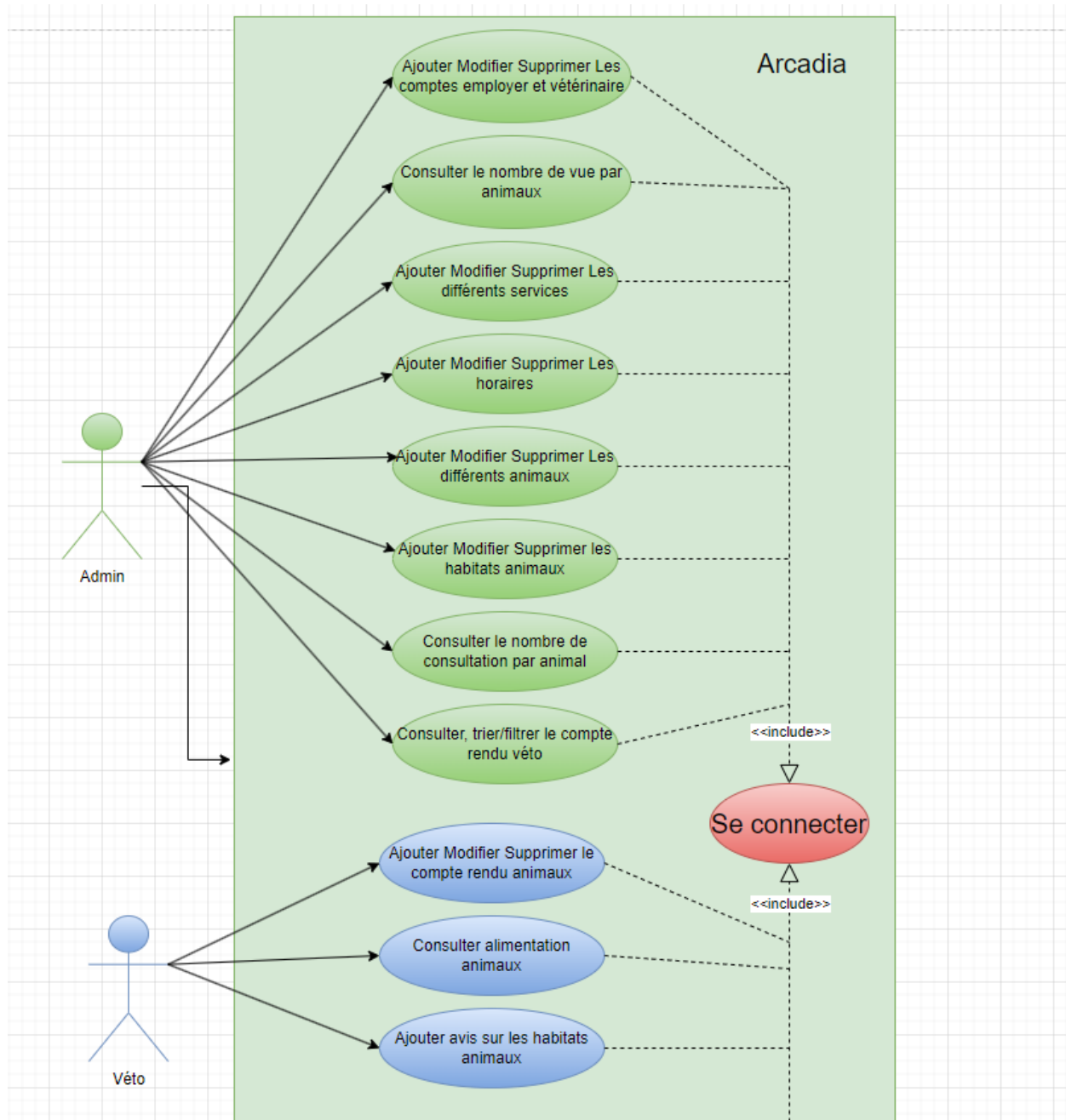
{# Fin section Horaires #}

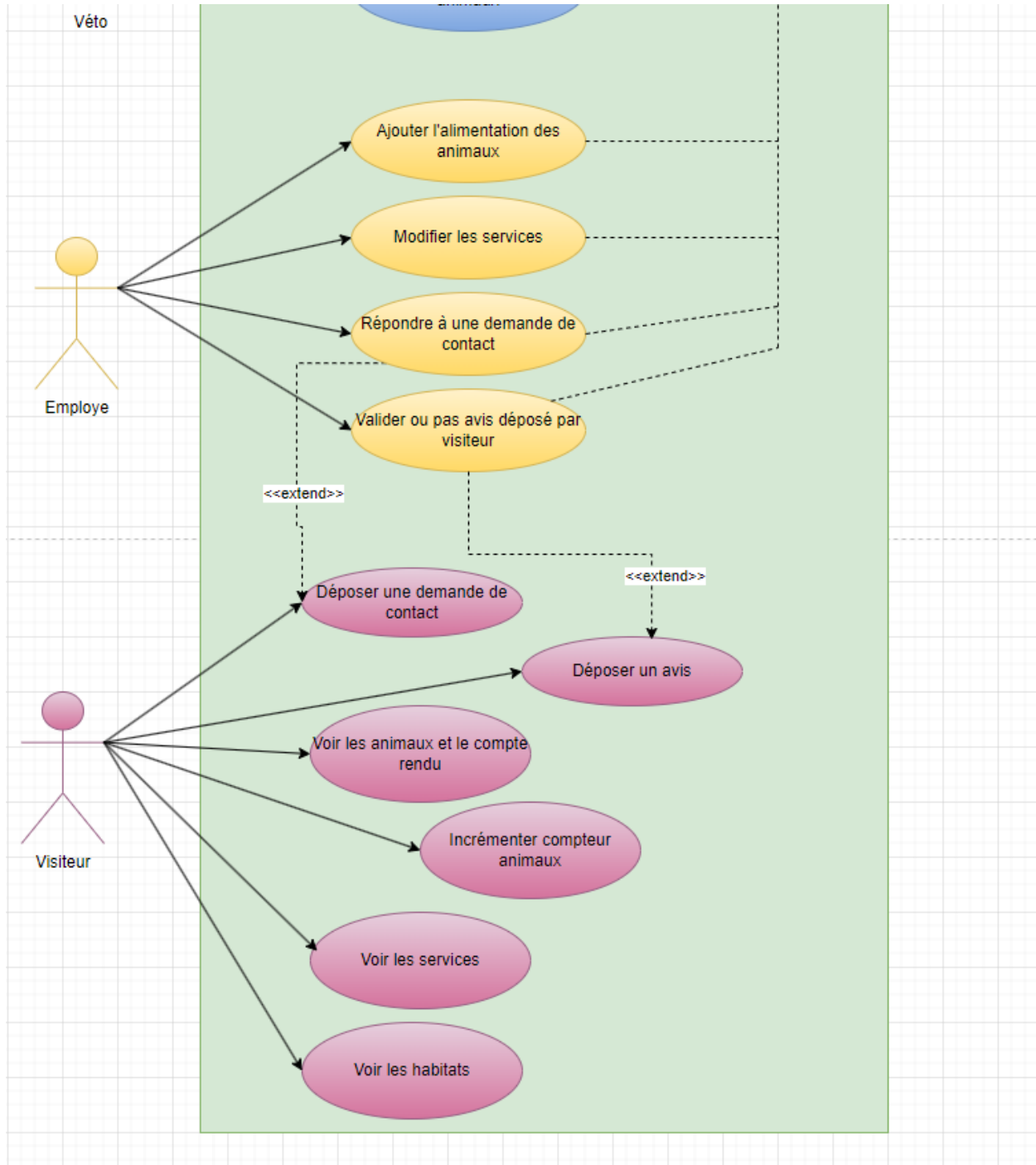
```

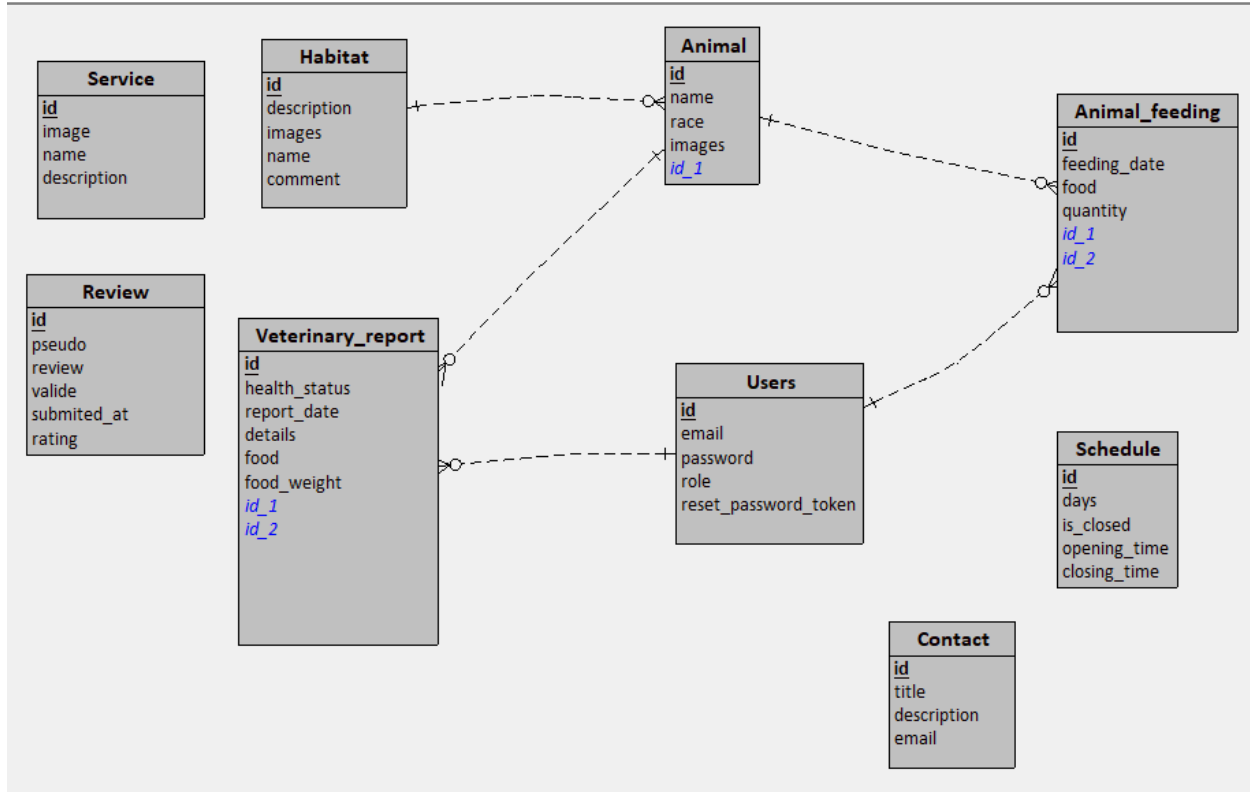
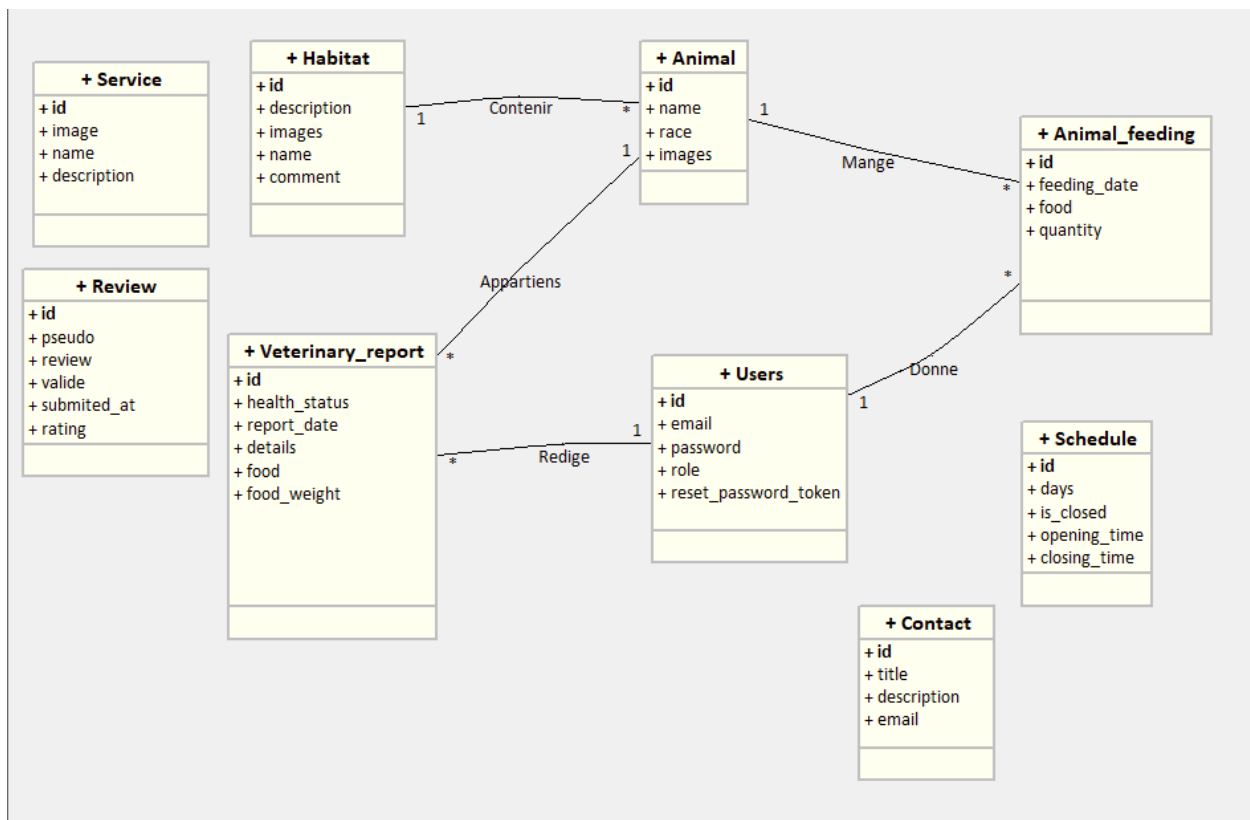
You, 24/08/2024 10:23 • Uncommitted changes



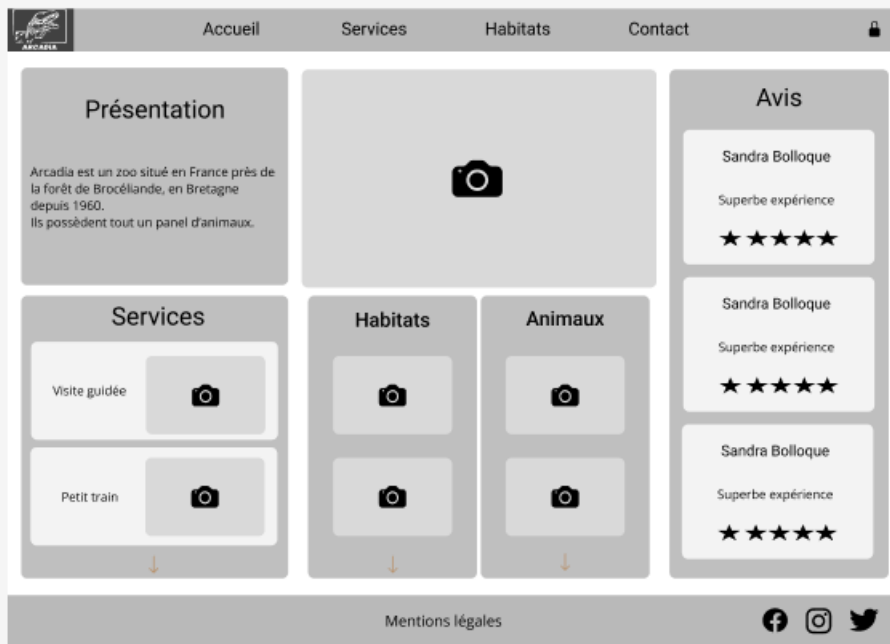




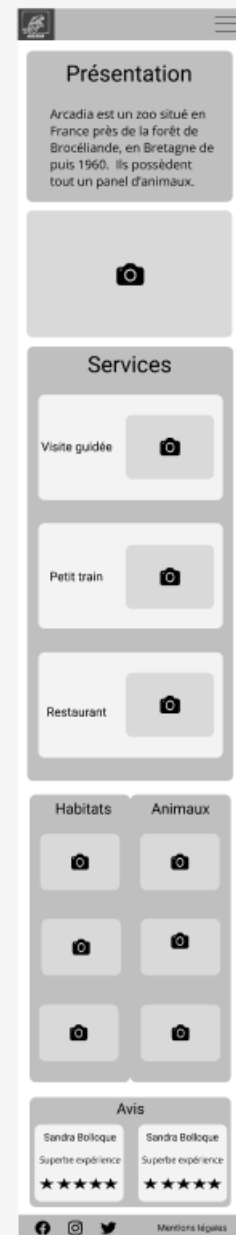


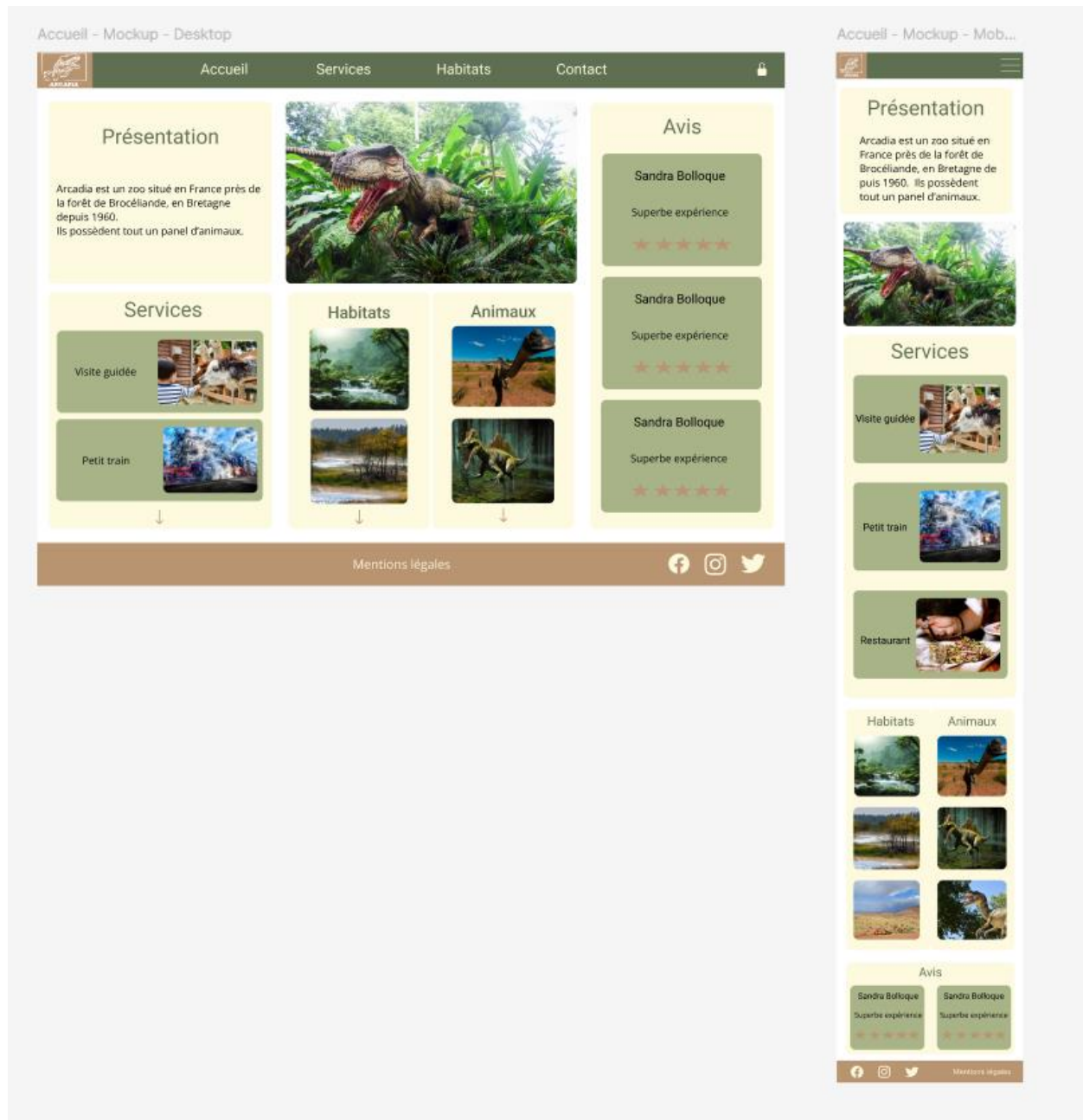


Accueil - Wireframe - Desktop



Accueil - Mockup - Mob...





```
1  USE arcadia_db;
2
3  CREATE TABLE IF NOT EXISTS habitat (
4      id INT AUTO_INCREMENT PRIMARY KEY,
5      name VARCHAR(100) NOT NULL,
6      description TEXT NOT NULL ,
7      image JSON NOT NULL ,
8      comment TEXT
9  );
10
11 CREATE TABLE IF NOT EXISTS animal (
12     id INT AUTO_INCREMENT PRIMARY KEY,
13     name VARCHAR(150) NOT NULL,
14     race VARCHAR(150) NOT NULL,
15     image JSON,
16     habitat_id INT
17     # FOREIGN KEY (habitat_id) REFERENCES habitat(id)
18 );
19
20 CREATE TABLE IF NOT EXISTS contact (
21     id INT AUTO_INCREMENT PRIMARY KEY,
22     title VARCHAR(255) NOT NULL,
23     description TEXT NOT NULL,
24     email VARCHAR(255) NOT NULL
25 );
26
27 CREATE TABLE IF NOT EXISTS service (
28     id INT AUTO_INCREMENT PRIMARY KEY,
29     name VARCHAR(100) NOT NULL,
30     description TEXT,
31     image JSON
32 );
33
34 CREATE TABLE IF NOT EXISTS review (
35     id INT AUTO_INCREMENT PRIMARY KEY,
36     pseudo VARCHAR(30) NOT NULL,
37     comment TEXT NOT NULL,
38     valid BOOLEAN DEFAULT FALSE,
39     submitted_at DATETIME NOT NULL,
40     rating INT NOT NULL
41 );
```

```
1  USE arcadia_db;
2
3  ALTER TABLE animal
4      ADD CONSTRAINT fk_animal_habitat
5          FOREIGN KEY (habitat_id) REFERENCES habitat(id)
6              ON DELETE SET NULL
7              ON UPDATE CASCADE;
8
9  ALTER TABLE veterinary_report
10     ADD CONSTRAINT fk_vet_report_animal
11         FOREIGN KEY (animal_id) REFERENCES animal(id)
12             ON DELETE CASCADE
13             ON UPDATE CASCADE;
14
15  ALTER TABLE veterinary_report
16     ADD CONSTRAINT fk_vet_report_user
17         FOREIGN KEY (user_id) REFERENCES user(id)
18             ON DELETE SET NULL
19             ON UPDATE CASCADE;
20
21  ALTER TABLE animal_feeding
22     ADD CONSTRAINT fk_animal_feeding_animal
23         FOREIGN KEY (animal_id) REFERENCES animal(id)
24             ON DELETE CASCADE
25             ON UPDATE CASCADE;
26
27  ALTER TABLE animal_feeding
28     ADD CONSTRAINT fk_animal_feeding_user
29         FOREIGN KEY (user_id) REFERENCES user(id)
30             ON DELETE SET NULL
31             ON UPDATE CASCADE;
```

```

1  USE arcadia_db;
2  SET NAMES 'utf8mb4';
3
4  INSERT INTO schedule
5      (id, day, opening_time, closing_time, is_closed)
6  VALUES
7      (1, 'Lundi', '09:00:00', '19:00:00', 0),
8      (3, 'Mardi', '09:00:00', '19:00:00', 0),
9      (4, 'Mercredi', '09:00:00', '19:00:00', 0),
10     (5, 'Jeudi', '09:00:00', '19:00:00', 0),
11     (6, 'Vendredi', '09:00:00', '19:00:00', 0),
12     (7, 'Samedi', '09:00:00', '19:00:00', 0),
13     (22, 'Dimanche', '09:00:00', '19:00:00', 1);
14
15  INSERT INTO 'habitat'
16      ('id', 'name', 'description', 'image', 'comment')
17  VALUES
18      (2, 'Savane', 'La savane, ça vient de l\'espagnol et ça veut dire « plaine sans arbres ». En fait c
19      (11, 'Marais', 'La tourbière est un écosystème\r\nconstamment saturé d\'eau au sein duquel\r\ns\'accu
20      (12, 'Jungle', 'La jungle est une immense forêt où poussent de façon très serrée, arbres, broussail
21
22  INSERT INTO 'service'
23      ('id', 'name', 'description', 'image')
24  VALUES
25      (6, 'Petit train', 'Découvrez notre petit train touristique, une aventure de 60 minutes à travers l
26      (7, 'Visite Guidée', 'Découvrez notre visite guidée au zoo, une immersion de 90 minutes avec un gui
27      (8, 'Restaurant', 'Savourez une expérience culinaire unique dans notre restaurant, où chaque plat e
28
29  INSERT INTO 'animal'
30      ('id', 'habitat_id', 'name', 'race', 'image')
31  VALUES
32      (8, 2, 'Wivine', 'Vélociraptor', '["velociraptor-66911c2e52919.jpg"]'),
33      (11, 12, 'Seb', 'Diplodocus', '["diplodocus-6693e96007d6d.jpg"]'),
34      (12, 2, 'Capt', 'Spinosauve', '["spinosaurus-6693f36737411.jpg"]'),
35      (13, 2, 'Exploris', 'T-Rex', '["1526479615762038050-669413357050c.jpg"]');| mrleoufff, 19/07/202
36
37  INSERT INTO 'veterinary_report'
38      ('id', 'animal_id', 'user_id', 'health_status', 'food', 'food_weight', 'report_date', 'detail')
39  VALUES
40      (1, 8, 48, 'Bonne santé', 'Chien', 'BF', '2021-07-14 09:00:00', 'Test positif')

```

```

1  {% extends 'base.html.twig' %} MrLeoufff, 09/07/2024 17:54 • affichage des avis
2
3  {% block body %}
4      <div class="container mt-5">
5          <div class="row">
6              <div class="col-md-12">
7                  <div class="card mb-4">
8                      <div class="card-body">
9                          <h1 class="card-title"><strong>{{ animal.name }}</strong></h1>
10                         <h2 class="card-text">Espèce: {{ animal.race }}</h2>
11                         <p>{{ animal.habitat.name }}</p>
12                         <div>
13                             <h3>Images</h3>
14                             {% for image in animal.image %}
15                                 
16                             {% endfor %}
17                         </div>
18                         <p class="card-text"><strong>Nombre de vues:</strong> {{ views }}</p>
19                     </div>
20                 </div>
21
22                 <div class="card mb-4">
23                     <div class="card-body">
24                         <h2 class="card-title">Rapports Vétérinaires</h2>
25                         {% if veterinaryReports is not empty %}
26                             <ul class="list-group">
27                                 {% for report in veterinaryReports %}
28                                     <li class="list-group-item">
29                                         <p><strong>Date:</strong> {{ report.reportDate|date('d/m/Y') }}</p>
30                                         <p><strong>État de santé:</strong> {{ report.healthStatus }}</p>
31                                         <p><strong>Nourriture:</strong> {{ report.food }}</p>
32                                         <p><strong>Poids de la nourriture:</strong> {{ report.foodWeight }} kg</p>
33                                         <p><strong>Détails:</strong> {{ report.detail }}</p>
34                                     </li>
35                                 {% endfor %}
36                             </ul>
37                         {% else %}
38                             <p>Aucun rapport vétérinaire disponible pour cet animal.</p>
39                         {% endif %}

```

```

jobs:
  symfony-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: shivammathur/setup-php@2cb9b829437ee246e9b3cac53555a39208ca6d28
        with:
          php-version: '8.3'
      - uses: actions/checkout@v4
      - name: Copy .env.test.local
        run: php -r "file_exists('.env.test.local') || copy('.env.test', '.env.test.local');"
      - name: Cache Composer packages
        id: composer-cache
        uses: actions/cache@v3
        with:
          path: vendor
          key: ${{ runner.os }}-php-${{ hashFiles('**/composer.lock') }}
          restore-keys: |
            ${{ runner.os }}-php-
      - name: Install Dependencies
        run: composer install -q --no-ansi --no-interaction --no-scripts --no-progress --prefer-dist
      - name: Create Database
        run: |
          mkdir -p data
          touch data/database.sqlite
      - name: Clear Symfony cache
        run: php bin/console cache:clear --env=test
      - name: Display Symfony logs
        run: tail -n 50 var/log/test.log
      - name: Start Symfony server
        run: php -S 0.0.0.0:8000 -t public/ > /dev/null 2>&1 &
      - name: Execute tests (Unit and Feature tests) via PHPUnit
        env:
          DATABASE_URL: sqlite:///kernel.project_dir%/data/database.sqlite
        run: vendor/bin/phpunit

```

```
1 | name: CD mrleoufff, 18/07/2024 17:23 • first CD
2 |
3 | push:
4 |   branches:
5 |     - main
6 |
7 | jobs:
8 |   deploy:
9 |     runs-on: ubuntu-latest
10 |
11 |     env:
12 |       SSHPASS: ${ secrets.SSHPASS }
13 |
14 |     steps:
15 |       - name: Checkout repository
16 |         uses: actions/checkout@v2
17 |
18 |       - name: Install sshpass & update apt
19 |         run: |
20 |           sudo apt update
21 |           sudo apt install sshpass
22 |
23 |       - name: Deploy application
24 |         run: |
25 |           sshpass -e ssh -o strictHostKeyChecking=no ${ secrets.SSH_USER }@${ secrets.SSH_IP } "
26 |             cd htdocs &&
27 |             cd reneleliard.online/ &&
28 |             git pull https://github.com/MrLeoufff/ecf-arcadia.git &&
29 |             composer install
30 |           "
```