



## Dossier Projet

Titre RNCP – Développeur Web / Web Mobile

Kreir Eliess

Studi – Promotion Mars/Avril 2025

Lien du site web : <https://arcadia.press/>

Lien du dépôt github : <https://github.com/EKreir/dossierProjet>

## Table des matières

Page 3 => Introduction

Page 5 => Architecture du projet

Page 25 => Design

Page 31 => Base de données

Page 47 => Développement backend

Page 60 => Cycle de vie d'une fonctionnalité

Page 62 => Développement frontend

Page 66 => Déploiement

Page 69 => Sécurité

Page 72 => Problèmes rencontrés et solutions

Page 75 => Annexes

# Introduction

## 1. Contexte du projet :

Ce projet vise à fournir une application web et mobile intuitive pour les visiteurs d'un parc animalier, leur permettant de découvrir les animaux, leurs habitats et les divers services proposés par le parc. L'application vise à offrir une expérience utilisateur fluide et interactive, permettant aux visiteurs de s'informer facilement sur les animaux et les installations du parc.

Parallèlement, l'application intègre des fonctionnalités spécifiques pour les administrateurs, employés et vétérinaires du parc, afin de faciliter la gestion des services, des avis des visiteurs, ainsi que le suivi des animaux et de leur état de santé. L'objectif est d'assurer une gestion efficace et un suivi optimal tout en garantissant la sécurité et l'accessibilité de l'ensemble des données. Ce projet a été conçu pour offrir une expérience personnalisée tant pour les visiteurs que pour le personnel, en alliant une interface conviviale à des outils de gestion performants.

## 2. Fonctionnalités principales :

- **Page d'accueil :**

Présenter le parc, ses valeurs écologiques, ainsi qu'un aperçu des animaux et de leurs habitats, afin de captiver l'attention des visiteurs dès leur arrivée sur le site.

- **Barre de navigation :**

Offrir une navigation fluide et intuitive, permettant un accès rapide et direct aux différentes sections principales de l'application (Accueil, Services, Habitats, Avis, Contact).

- **Découverte des habitats et des animaux :**

Permettre aux utilisateurs de consulter des informations sur les animaux du parc, leurs habitats, ainsi que leur état de santé, favorisant une expérience d'apprentissage enrichissante.

- **Services :**

Informier les visiteurs sur les différents services proposés par le parc (restauration, visites guidées, etc.), avec la possibilité de consulter les horaires disponibles.

- **Avis des visiteurs :**

Offrir aux visiteurs la possibilité de laisser des commentaires sur leur expérience au parc. Ces avis seront soumis à validation par le personnel avant d'être publiés.

- **Espace Administrateur :**

Permettre aux administrateurs de gérer les comptes utilisateurs, d'administrer les services, de mettre à jour les informations sur les

habitats et les animaux, ainsi que de consulter les rapports vétérinaires relatifs à l'état de santé des animaux.

- **Espace Employé :**

Fournir aux employés les outils nécessaires pour valider les avis des visiteurs, gérer les services en cours et assurer le bon fonctionnement quotidien du parc.

- **Espace Vétérinaire :**

Permettre aux vétérinaires d'accéder aux informations sur la santé des animaux, de mettre à jour leurs fiches médicales et de suivre l'évolution de leur état de santé.

- **Connexion sécurisée :**

Assurer une connexion sécurisée pour les utilisateurs ayant un accès restreint (administrateurs, employés, vétérinaires), via un système d'identifiants sécurisés et de gestion des rôles.

### 3. Technologies utilisées :

Partie frontend :



HTML5 pour structurer le contenu de mon site web



CSS3 pour la mise en forme et du style du site web



Javascript pour rendre les pages web dynamiques



Bootstrap pour la structure interface utilisateurs déjà préparée

Partie backend :



PHP pour développer le script du site côté serveur



Mailpit pour la partie gestion des e-mails



Composer pour installer des bibliothèques dont le projet a besoin

Partie base de données :



MySQL pour le développement de la base de données relationnelle



MongoDB uniquement pour la partie consultation animale. Données accessibles uniquement en lecture/écriture de manière régulière

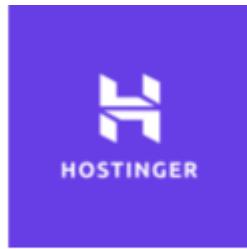


phpMyAdmin pour l'interface de la gestion des bases de données

Partie développement, tests et déploiement :



Visual Studio Code (vs code),  
l'IDE utilisé pour le développement  
du projet



Hostinger l'hébergeur  
où est déployé le projet



MongoDB Atlas pour la gestion de la base  
de données non-relational NoSQL dans  
le cloud

Partie conception et organisation :



Notion pour la gestion du  
projet et tâches



Lucidchart pour les diagrammes  
user case, UML, ...



Dbdiagram.io pour le  
diagramme MLD



Figma pour les  
maquettes et  
wireframes

## Architecture du projet

- **Structure des dossiers selon le modèle MVC (Modèle-Vue-Contrôleur) :**

**Définition :** **MVC** est un acronyme pour **Modèle-Vue-Contrôleur** (Model-View-Controller en anglais). C'est un pattern de conception qui aide à organiser le code de manière claire et structurée. Voici ce que chaque composant représente :

- **Modèle (Model)** : C'est la partie qui gère la logique de l'application et les interactions avec la base de données. Il représente les données de l'application et les règles de gestion qui s'y rapportent. Par exemple, pour une application de gestion de livres, le modèle pourrait gérer les informations sur les livres (titre, auteur, etc.) et interagir avec la base de données pour récupérer ou enregistrer ces données.
- **Vue (View)** : C'est la partie qui représente l'interface utilisateur (UI). Elle est responsable de l'affichage des données que l'utilisateur voit. En PHP, la vue serait souvent des fichiers HTML ou des templates qui affichent les

informations du modèle sous une forme compréhensible pour l'utilisateur. Par exemple, afficher la liste des livres dans une page web.

- **Contrôleur (Controller)** : Le contrôleur est le médiateur entre le modèle et la vue. Il reçoit les requêtes de l'utilisateur (par exemple, cliquer sur un bouton ou soumettre un formulaire), traite cette action, interagit avec le modèle pour obtenir ou modifier les données nécessaires, puis transmet ces données à la vue pour affichage. Par exemple, un contrôleur pourrait récupérer une liste de livres à partir du modèle, puis afficher cette liste dans la vue.
- En résumé, l'architecture MVC permet de séparer les préoccupations dans une application web :
- **Modèle** : gestion des données.
- **Vue** : gestion de l'affichage.
- **Contrôleur** : gestion des actions de l'utilisateur et de la logique métier.

### Structure du projet :

> config	#Connexion à MySQL et MongoDB, gestion des routes
> controllers	#Tous les contrôleurs du projet
> info	#Informations utiles
> livrables	#Documents PDF ou autres livrables
> models	#Tous les modèles du projet
▼ public	
> css	#Tous les fichiers CSS
> images	#Toutes les images utilisées du site
> js	#Le fichier JavaScript
🐘 index.php	#Fichier d'entrée principal du site
> vendor	#Dossier généré par Composer pour les dépendances
▼ views	
> admin	#Vues spécifiques à l'espace administrateur
> employe	#Vues spécifiques à l'espace employé
> template	#Contient les éléments réutilisables(navbar, header, etc.)
> veto	#Vues spécifiques à l'espace vétérinaire
🐘 animal_details.php	#Le reste des vues pour la partie visiteurs
🐘 contact.php	#Fichiers et répertoires à ignorer par Git
🐘 habitats.php	
🐘 home.php	
🐘 legacy.php	
🐘 login.php	
🐘 logout.php	
🐘 policy.php	
🐘 services.php	

#Fichier SQL pour la structure de la base de données

#Dépendances et configurations de Composer

#Fichier de verrouillage des dépendances

- **Explication du code backend (PHP et MySQL) en suivant le modèle MVC**
  - Structure et rôle de chaque modèle :
    - **Animal.php** : Ce modèle gère toutes les informations relatives aux animaux, y compris leur état de santé, leur habitat, etc. Il interagit avec la base de données pour effectuer des opérations telles que la création, la mise à jour, la suppression et la récupération d'animaux.
    - **Contact.php** : Responsable de la gestion des messages envoyés par les visiteurs. Ce modèle traite l'envoi des emails et peut également gérer la réponse des employés.
    - **Service.php** : Gère les informations sur les services offerts par le parc (par exemple, visites guidées, restauration) et permet d'ajouter, supprimer ou modifier ces services.
    - **Habitat.php** : Ce modèle représente un habitat spécifique dans le parc, en stockant des informations comme le type d'habitat, les animaux associés, etc.
    - **ReviewModel.php** : Gère les avis laissés par les visiteurs sur le parc. Il permet de récupérer, ajouter, modifier et supprimer les avis.
    - **HabitatReview.php** : Ce modèle est associé à la gestion des commentaires des vétérinaires sur l'état de santé des animaux en lien avec leur habitat. Il permet de garder une trace des évaluations des vétérinaires sur chaque habitat.
    - **OpeningHours.php** : Ce modèle gère les horaires d'ouverture du zoo, permettant de mettre à jour les heures et d'afficher les informations correspondantes.
    - **FeedingModel.php** : Responsable de l'ajout des consommations alimentaires des animaux. Ce modèle permet d'ajouter et de suivre les repas donnés aux animaux.
    - **ReportModel.php** : Ce modèle est destiné à la gestion des rapports vétérinaires sur les animaux. Il permet aux vétérinaires de créer des rapports sur l'état de santé des animaux.
    - **User.php** : Gère la création des comptes pour les employés et les vétérinaires. Ce modèle permet d'ajouter, modifier et supprimer des utilisateurs, et de gérer leurs permissions et rôles.
  - Structure et rôle de chaque contrôleur :
    - **AdminController.php** : Dashboard et gestion des vues animales.
    - **AnimalController.php** : Création, modification, suppression et affichage des animaux.
    - **ContactController.php** : Gestion des mails visiteurs et des réponses.

- **FeedingController.php** : Suivi des consommations alimentaires des animaux.
- **HabitatController.php** : Gestion des habitats.
- **LoginController.php** : Gestion des connexions et redirections basées sur les rôles.
- **UserController.php** : Création et gestion des comptes utilisateurs (employés et vétérinaires).
- **ServiceController.php** : Gestion des services proposés par le parc.
- **ReviewController.php** : Gestion des avis des visiteurs.
- **ReportController.php** : Gestion des rapports vétérinaires.
- **OpeningHoursController.php** : Gestion des horaires du zoo.
- **PublicController.php** : Vues publiques pour les visiteurs (affichage des animaux, habitats, services).

Exemple de structure d'un modèle avec le modèle Animal :

\_\_construct() :

```
class Animal {
    private $conn;
    private $collection;

    public function __construct() {
        $database = new Database();
        $mongo = new MongoDBConnection();
        $this->conn = $database->getConnection();
        $this->collection = $mongo->getCollection('animal_views');
    }
}
```

**Rôle** : Le constructeur initialise les connexions à la base de données **MySQL** et à la collection **MongoDB** pour la gestion des vues des animaux.

**MySQL** : La connexion MySQL est obtenue via la classe **Database** (probablement une classe qui gère la connexion à MySQL).

**MongoDB** : La connexion MongoDB est obtenue via la classe **MongoDBConnection** pour accéder à la collection **animal\_views**, où les vues des animaux sont stockées.

`create($name, $breed, $image, $habitatId)` :

```

public function create($name, $breed, $image, $habitatId) {
    try {
        $query = "INSERT INTO animals (name, breed, image, habitat_id) VALUES (:name, :breed, :image, :habitat_id)";
        $stmt = $this->conn->prepare($query);

        $stmt->bindParam(':name', $name);
        $stmt->bindParam(':breed', $breed);
        $stmt->bindParam(':image', $image);
        $stmt->bindParam(':habitat_id', $habitatId);

        return $stmt->execute();
    } catch (Exception $e) {
        echo "Erreur lors de la création de l'animal : " . $e->getMessage();
        return false;
    }
}

```

**Rôle** : Crée un nouvel animal dans la base de données MySQL.

**MySQL** : Utilise une requête `INSERT INTO` pour insérer un nouvel enregistrement dans la table `animals` avec les données passées en paramètres (nom, race, image, et habitat).

**Retour** : Retourne `true` si l'exécution de la requête est réussie, sinon `false` en cas d'erreur.

`update($id, $name, $breed, $image, $habitatId)` :

```

public function update($id, $name, $breed, $image, $habitatId) {
    try {
        $query = "UPDATE animals SET name = :name, breed = :breed, image = :image, habitat_id = :habitat_id WHERE id = :id";
        $stmt = $this->conn->prepare($query);

        $stmt->bindParam(':id', $id);
        $stmt->bindParam(':name', $name);
        $stmt->bindParam(':breed', $breed);
        $stmt->bindParam(':image', $image);
        $stmt->bindParam(':habitat_id', $habitatId);

        return $stmt->execute();
    } catch (Exception $e) {
        echo "Erreur lors de la modification de l'animal : " . $e->getMessage();
        return false;
    }
}

```

**Rôle** : Met à jour les informations d'un animal existant dans la base de données.

**MySQL** : Effectue une requête `UPDATE` pour modifier les champs de l'animal (nom, race, image, habitat) en fonction de son `id`.

**Retour** : Retourne `true` si la mise à jour est réussie, sinon `false` en cas d'erreur.

`delete($id)` :

```

public function delete($id) {
    try {
        $query = "DELETE FROM animals WHERE id = :id";
        $stmt = $this->conn->prepare($query);
        $stmt->bindParam(':id', $id);
        return $stmt->execute();
    } catch (Exception $e) {
        echo "Erreur lors de la suppression de l'animal : " . $e->getMessage();
        return false;
    }
}

```

**Rôle** : Supprime un animal de la base de données.

**MySQL** : Utilise une requête `DELETE` pour supprimer un enregistrement dans la table `animals` en fonction de l'`id` passé en paramètre.

**Retour** : Retourne `true` si l'animal a été supprimé avec succès, sinon `false` en cas d'erreur.

`getAllAnimals()` :

```

public function getAllAnimals() {
    try {
        $query = "
SELECT a.id, a.name, a.breed, a.image, h.name AS habitat
FROM animals a
LEFT JOIN habitats h ON a.habitat_id = h.id";
        $stmt = $this->conn->prepare($query);
        $stmt->execute();
        $animals = $stmt->fetchAll(PDO::FETCH_ASSOC);

        foreach ($animals as &$animal) {
            $animal['views_count'] = $this->getAnimalViews($animal['id']);
        }
        return $animals;
    } catch (Exception $e) {
        echo "Erreur lors de la récupération des animaux : " . $e->getMessage();
        return [];
    }
}

```

**Rôle** : Récupère tous les animaux de la base de données, y compris leur habitat associé et leur nombre de vues.

**MySQL** : Exécute une requête `SELECT` pour obtenir les animaux et leurs habitats via une jointure `LEFT JOIN` entre les tables `animals` et `habitats`.

**MongoDB** : Appelle la méthode `getAnimalViews()` pour ajouter le nombre de vues depuis MongoDB pour chaque animal.

**Retour** : Un tableau associatif contenant tous les animaux avec leurs informations (nom, race, habitat, vues).

`getAllAnimal()` :

```
public function getAllAnimal() {
    $sql = "SELECT id, name FROM animals ORDER BY name ASC";
    $stmt = $this->conn->prepare($sql);
    $stmt->execute();
    return $stmt->fetchAll(PDO::FETCH_ASSOC);
}
```

**Rôle** : Récupère une liste simple d'animaux (id et nom), triée par nom.

**MySQL** : Effectue une requête `SELECT` pour obtenir une liste d'animaux, triée par nom dans l'ordre croissant.

**Retour** : Un tableau associatif contenant les ID et noms de tous les animaux.

`getAnimalById($id)` :

```
public function getAnimalById($id) {
    try {
        $query = "
            SELECT
                a.*,
                h.name AS habitat_name
            FROM animals a
            LEFT JOIN habitats h ON a.habitat_id = h.id
            WHERE a.id = :id
        ";
        $stmt = $this->conn->prepare($query);
        $stmt->bindParam(':id', $id);
        $stmt->execute();
        return $stmt->fetch(PDO::FETCH_ASSOC);
    } catch (Exception $e) {
        echo "Erreur lors de la récupération de l'animal : " . $e->getMessage();
        return null;
    }
}
```

**Rôle** : Récupère les détails d'un animal spécifique en fonction de son `id`.

**MySQL** : Utilise une requête `SELECT` pour obtenir toutes les informations de l'animal ainsi que le nom de son habitat via une jointure `LEFT JOIN`.

**Retour** : Un tableau associatif contenant toutes les informations de l'animal et de son habitat.

`getHabitatOptions()` :

```

public function getHabitatOptions() {
    try {
        $query = "SELECT * FROM habitats";
        $stmt = $this->conn->prepare($query);
        $stmt->execute();
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    } catch (Exception $e) {
        echo "Erreur lors de la récupération des habitats : " . $e->getMessage();
        return [];
    }
}

```

**Rôle** : Récupère tous les habitats disponibles pour les associer aux animaux.

**MySQL** : Effectue une requête `SELECT` pour obtenir tous les enregistrements de la table `habitats`.

**Retour** : Un tableau associatif contenant tous les habitats.

getAnimalsByHabitat(\$habitatId) :

```

public function getAnimalsByHabitat($habitatId) {
    try {
        $query = "SELECT * FROM animals WHERE habitat_id = :habitat_id";
        $stmt = $this->conn->prepare($query);
        $stmt->bindParam(':habitat_id', $habitatId);
        $stmt->execute();
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    } catch (Exception $e) {
        echo "Erreur lors de la récupération des animaux par habitat : " . $e->getMessage();
        return [];
    }
}

```

**Rôle** : Récupère tous les animaux d'un habitat spécifique.

**MySQL** : Effectue une requête `SELECT` pour obtenir les animaux dont l'`habitat_id` correspond à celui passé en paramètre.

**Retour** : Un tableau associatif contenant les animaux de l'habitat spécifié.

getAnimalViews(\$animalId) :

```

public function getAnimalViews($animalId) {
    $viewData = $this->collection->findOne(['animal_id' => (int)$animalId]);

    return $viewData ? $viewData['views_count'] : 0;
}

```

**Rôle** : Récupère le nombre de vues d'un animal à partir de la collection MongoDB.

**MongoDB** : Utilise une requête MongoDB pour récupérer les vues de l'animal basé sur son `animal_id`. Si les vues sont trouvées, elles sont renvoyées.

**Retour** : Le nombre de vues de l'animal (ou `0` si aucune donnée n'est trouvée).

`getAnimalMongo()` :

```
public function getAnimalMongo() {
    // Effectuer une requête pour récupérer tous les animaux de la collection
    $animals = $this->collection->find()->toArray(); // Utilisation de *****

    return $animals; // Retourner les animaux
}
```

**Rôle** : Récupère tous les animaux stockés dans MongoDB.

**MongoDB** : Effectue une requête MongoDB pour récupérer tous les documents de la collection `animal_views` et les retourne sous forme de tableau.

**Retour** : Un tableau contenant tous les animaux de MongoDB.

Exemple de structure d'un contrôleur avec le contrôleur `AnimalController` :

`__construct()` :

```
class AnimalController {
    private $animalModel;

    public function __construct() {
        $this->animalModel = new Animal();
    }
}
```

**Rôle** : Le constructeur initialise le modèle `Animal` en tant qu'objet privé. Ce modèle est ensuite utilisé dans toutes les méthodes du contrôleur pour interagir avec la base de données ou MongoDB.

**Action** : L'initialisation du modèle `Animal` permet d'utiliser ses méthodes pour gérer les données des animaux.

`listAnimals()` :

```
public function listAnimals() {
    $animals = $this->animalModel->getAllAnimals();
    require_once __DIR__ . '/..views/admin/list_animals.php';
}
```

**Rôle :** Cette méthode est responsable de l'affichage de la liste des animaux dans l'interface d'administration.

### Fonctionnalité :

- Appelle la méthode `getAllAnimals()` du modèle `Animal` pour récupérer tous les animaux présents dans la base de données.
- Passe ces animaux à la vue `list_animals.php`, qui est responsable de l'affichage de ces données dans l'interface utilisateur (admin).

`createAnimal()` :

```
public function createAnimal() {
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $name = trim($_POST['name']);
        $breed = trim($_POST['breed']);
        $habitat_id = (int) $_POST['habitat_id'];

        if (empty($name) || empty($breed) || empty($habitat_id)) {
            $errorMessage = "Tous les champs sont obligatoires.";
            require_once __DIR__ . '/..views/admin/create_animal.php';
            return;
        }

        // Gérer l'upload de l'image
        $imagePath = '';
        if (isset($_FILES['image']) && $_FILES['image']['error'] === 0) {
            $imagePath = '/images/' . basename($_FILES['image']['name']);
            move_uploaded_file($_FILES['image']['tmp_name'], __DIR__ . '/../public' . $imagePath);
        }

        // Ajouter l'animal dans la base de données
        $success = $this->animalModel->create($name, $breed, $imagePath, $habitat_id);

        if ($success) {
            $successMessage = "Animal créé avec succès !";
        } else {
            $errorMessage = "Une erreur est survenue lors de la création de l'animal.";
        }

        require_once __DIR__ . '/..views/admin/create_animal.php';
    } else {
        $habitats = $this->animalModel->getHabitatOptions();
        require_once __DIR__ . '/..views/admin/create_animal.php';
    }
}
```

**Rôle :** Cette méthode permet de créer un nouvel animal dans la base de données.

### Fonctionnalité :

- Si la méthode HTTP est **POST**, elle récupère les données envoyées par le formulaire, comme le nom, la race, et l'habitat de l'animal.
- Elle vérifie que tous les champs sont remplis, sinon elle affiche un message d'erreur.
- Elle gère l'upload de l'image associée à l'animal et la déplace dans le dossier **/public/images/**.
- Ensuite, elle appelle la méthode **create()** du modèle **Animal** pour insérer l'animal dans la base de données.
- En fonction de la réussite de l'opération, elle affiche un message de succès ou d'erreur, puis recharge la vue **create\_animal.php**.
- Si la méthode HTTP est différente de **POST**, elle récupère les habitats disponibles pour l'animal via **getHabitatOptions()** et charge la vue **create\_animal.php**.

`editAnimal()` :

```
public function editAnimal() {
    $id = $_GET['id'] ?? null;

    if (!$id) {
        echo "Animal introuvable.";
        return;
    }

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $name = trim($_POST['name']);
        $breed = trim($_POST['breed']);
        $habitat_id = (int) $_POST['habitat_id'];

        if (empty($name) || empty($breed) || empty($habitat_id)) {
            $errorMessage = "Tous les champs sont obligatoires.";
            require_once __DIR__ . '/../../views/admin/edit_animal.php';
            return;
        }

        // Gérer l'upload de l'image
        $imagePath = '';
        if (isset($_FILES['image']) && $_FILES['image']['error'] === 0) {
            $imagePath = '/images/' . basename($_FILES['image']['name']);
            move_uploaded_file($_FILES['image']['tmp_name'], __DIR__ . '/../../public' . $imagePath);
        }
    }
}
```

```

    // Mettre à jour l'animal
    $success = $this->animalModel->update($id, $name, $breed, $imagePath, $habitat_id);

    if ($success) {
        $successMessage = "Animal modifié avec succès !";
    } else {
        $errorMessage = "Une erreur est survenue lors de la modification de l'animal.";
    }

    require_once __DIR__ . '/../../../../views/admin/edit_animal.php';
} else {
    $animal = $this->animalModel->getAnimalById($id);
    $habitats = $this->animalModel->getHabitatOptions();
    require_once __DIR__ . '/../../../../views/admin/edit_animal.php';
}
}

```

**Rôle :** Cette méthode permet de modifier un animal existant.

**Fonctionnalité :**

- Si un identifiant (`id`) est passé en paramètre via `GET`, elle vérifie que l'animal existe et charge les informations actuelles de cet animal via `getAnimalById()`.
- Si la méthode HTTP est `POST`, elle récupère les nouvelles informations pour l'animal (nom, race, habitat, image).
- Comme pour la création d'un animal, elle gère l'upload de l'image, met à jour l'animal via la méthode `update()` du modèle `Animal`, et affiche un message de succès ou d'erreur dans la vue `edit_animal.php`.
- Si la méthode HTTP est différente de `POST`, elle récupère les informations de l'animal à modifier et les habitats disponibles avant d'afficher le formulaire d'édition.

`deleteAnimal()` :

```

public function deleteAnimal() {
    $id = $_GET['id'] ?? null;

    if (!$id) {
        echo "Animal introuvable.";
        return;
    }

    $success = $this->animalModel->delete($id);

    if ($success) {
        header('Location: /list-animals');
        exit;
    } else {
        echo "Une erreur est survenue lors de la suppression de l'animal.";
    }
}

```

**Rôle :** Cette méthode permet de supprimer un animal de la base de données.

## Fonctionnalité :

- Elle récupère l'identifiant de l'animal à supprimer via GET.
- Si l'identifiant est valide, elle appelle la méthode `delete()` du modèle `Animal` pour supprimer l'animal de la base de données.
- En cas de succès, elle redirige vers la page de liste des animaux (`/list-animals`), sinon elle affiche un message d'erreur.

### showAnimalDetails() :

```

public function showAnimalDetails() {
    $id = $_GET['id'] ?? null;

    if (!$id) {
        echo "Identifiant de l'animal manquant.";
        return;
    }

    $animal = $this->animalModel->getAnimalById($id);

    if (!$animal) {
        echo "Animal introuvable.";
        return;
    }

    // Ajouter ou mettre à jour le compteur de consultations dans MongoDB
    $mongo = new MongoDBConnection();
    $collection = $mongo->getCollection('animal_views');

    // Chercher si un document existe déjà pour cet animal
    $existingView = $collection->findOne(['animal_id' => (int)$id]);

    if ($existingView) {
        // Si le document existe, mettre à jour le compteur
        $collection->updateOne(
            ['animal_id' => (int)$id],
            ['$inc' => ['views_count' => 1]]
        );
    } else {
        // Si le document n'existe pas, créer un nouveau document
        $collection->insertOne([
            'animal_id' => (int)$id,
            'views_count' => 1
        ]);
    }

    $reportModel = new ReportModel();
    $report = $reportModel->getLastReportByAnimalId($id);

    // Passe les données à la vue
    require_once __DIR__ . '/../views/animal_details.php';
}

```

**Rôle :** Afficher les détails d'un animal spécifique, y compris son nombre de vues.

**Fonctionnalité :**

- Elle récupère l'identifiant de l'animal via `GET`, puis appelle la méthode `getAnimalById()` du modèle `Animal` pour obtenir les informations détaillées de l'animal.
- Ensuite, elle interagit avec MongoDB via `MongoDBConnection` pour incrémenter ou initialiser le compteur de vues de l'animal dans la collection `animal_views`.
- Si l'animal existe, elle passe les informations de l'animal et son rapport vétérinaire (via le modèle `ReportModel`) à la vue `animal_details.php` pour l'affichage.
- Si l'animal n'existe pas, un message d'erreur est affiché.

`showAnimalCount()` :

```
public function showAnimalCount() {  
    // Récupérer tous les animaux avec leur nombre de vues  
    $animals = $this->animalModel->getAllAnimals();  
    // Passer les données à la vue  
    require_once __DIR__ . '/../../views/admin/animal_count.php';  
}
```

**Rôle :** Afficher tous les animaux avec leur nombre de vues.

**Fonctionnalité :**

- Elle appelle la méthode `getAllAnimals()` du modèle `Animal` pour récupérer la liste de tous les animaux avec leurs informations et leurs statistiques de vues.
- Ensuite, elle passe ces données à la vue `animal_count.php` pour afficher le nombre de vues de chaque animal.

**Résumé des Méthodes :**

- `listAnimals()` : Liste tous les animaux dans l'interface admin.
- `createAnimal()` : Crée un nouvel animal en validant les données et en gérant l'upload d'image.
- `editAnimal()` : Modifie les informations d'un animal existant et gère l'upload d'image.
- `deleteAnimal()` : Supprime un animal de la base de données.

- **showAnimalDetails()** : Affiche les détails d'un animal, met à jour le nombre de vues et affiche un rapport vétérinaire.
- **showAnimalCount()** : Affiche la liste des animaux avec leur nombre de vues.

/config/Database.php :

```
<?php

use PDO;

class Database {
    private $host = 'localhost';
    private $dName = '*****';
    private $username = '*****';
    private $password = '*****';
    public $conn;

    public function getConnection() {
        $this->conn = null;

        try {
            $this->conn = new PDO("mysql:host=$this->host;dbname=$this->dName", $this->username, $this->password);
            $this->conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        } catch (PDOException $exception) {
            echo "Erreur de connexion : " . $exception->getMessage();
        }

        return $this->conn;
    }
}
```

Ce fichier contient la classe **Database**, qui permet de gérer la connexion à la base de données MySQL. Elle utilise **PDO** (PHP Data Objects), une méthode plus sécurisée et flexible pour interagir avec les bases de données dans PHP.

## 1. Déclaration des variables privées

- **\$host** : Contient l'adresse du serveur de base de données, ici configuré à **localhost** (ce qui signifie que la base de données est hébergée localement sur le même serveur que l'application).
- **\$dName** : Contient le nom de la base de données à laquelle se connecter.
- **\$username** : Le nom d'utilisateur utilisé pour se connecter à la base de données.
- **\$password** : Le mot de passe de l'utilisateur MySQL.
- **\$conn** : Cette variable représente la connexion PDO qui sera utilisée pour interagir avec la base de données.

## 2. Méthode **getConnection()**

- **Rôle** : Cette méthode établit une connexion à la base de données MySQL et renvoie l'objet de connexion PDO.
- **Fonctionnalité** :
  - D'abord, la connexion est initialisée à **null** pour garantir qu'une ancienne connexion ne persiste pas.

- Ensuite, elle tente d'établir une connexion à la base de données à l'aide de **PDO** avec les informations de connexion : `host`, `dbname`, `username`, et `password`.
- La méthode **setAttribute()** est utilisée pour définir un attribut de l'objet PDO, ici pour activer le mode d'erreur **ERRMODE\_EXCEPTION** afin que PDO génère une exception en cas d'erreur. Cela permet de capturer les erreurs de manière propre et contrôlée.
- Si une exception est lancée (par exemple, si la connexion échoue), elle est capturée via un **try-catch** et un message d'erreur est affiché.
- Si la connexion réussit, l'objet de connexion est retourné pour être utilisé dans d'autres parties de l'application.

### 3. Retour de la connexion

- À la fin de la méthode, si la connexion est établie avec succès, l'objet de connexion `$this->conn` est renvoyé.
- Cet objet pourra ensuite être utilisé pour exécuter des requêtes SQL, interagir avec la base de données et récupérer des données dans les autres classes ou contrôleurs du projet.

/config/MongoDB.php :

```
<?php

require '../vendor/autoload.php';

class MongoDBConnection {
    private $client;
    private $db;

    public function __construct() {
        // Connexion à MongoDB
        $this->client = new MongoDB\Client("mongodb://localhost:27017"); // Assurez-vous que le port et l'URL sont
corrects
        $this->db = $this->client->zoo; // Base de données 'zoo'
    }

    public function getCollection($collectionName) {
        return $this->db->$collectionName;
    }
}
```

Ce fichier contient la classe **MongoDBConnection**, qui permet de gérer la connexion à la base de données MongoDB et d'interagir avec des collections spécifiques de cette base.

#### 1. Chargement des dépendances

- `require '../vendor/autoload.php';` : Cette ligne inclut le fichier `autoload.php` généré par **Composer** pour charger toutes les dépendances nécessaires au projet, y compris le client MongoDB. Cela garantit que toutes les classes et bibliothèques utilisées dans le fichier sont automatiquement chargées.

## 2. Déclaration des variables privées

- **\$client** : Cette variable représente une instance de **MongoDB\Client** qui est utilisée pour se connecter à MongoDB. Elle est initialisée dans le constructeur et permet d'établir une connexion avec la base de données MongoDB.
- **\$db** : Cette variable représente la base de données MongoDB à laquelle nous souhaitons accéder. Elle est définie à **zoo**, qui est la base de données spécifique pour ce projet. Si la base de données n'existe pas, MongoDB la crée automatiquement lorsqu'une opération est effectuée sur cette base.

## 3. Méthode `__construct()`

- **Rôle** : Le constructeur établit la connexion à MongoDB dès qu'une instance de la classe **MongoDBConnection** est créée.
- **Fonctionnalité** :

- Il initialise l'instance **\$client** en créant un objet **MongoDB\Client**, avec l'URL de connexion "**mongodb://localhost:27017**", qui est le point d'entrée standard pour une instance MongoDB locale. Si MongoDB est hébergé à un autre endroit (par exemple, sur un cloud ou un serveur distant), cette URL doit être mise à jour.
- Il initialise l'instance **\$db** avec la base de données nommée **zoo**. Si cette base de données n'existe pas encore, MongoDB la crée automatiquement dès qu'une collection ou un document est ajouté à cette base.

## 4. Méthode `getCollection($collectionName)`

- **Rôle** : Cette méthode permet d'obtenir une collection spécifique de la base de données **zoo**.
- **Fonctionnalité** :
  - Elle prend un paramètre **\$collectionName**, qui correspond au nom de la collection MongoDB que nous souhaitons récupérer.
  - Elle retourne la collection spécifiée de la base de données **\$this->db**. Par exemple, si le paramètre **\$collectionName** est **animal\_views**, la méthode retournera la collection **animal\_views** de la base de données **zoo**.
  - Cela permet à d'autres classes ou modèles d'interagir facilement avec les collections MongoDB sans avoir à se préoccuper de la logique de connexion.

/config/Router.php :

```
<?php

class Router {
    private $routes = [];

    // Ajouter une route
    public function add($path, $callback) {
        $this->routes[$path] = $callback;
    }

    // Lancer le routeur
    public function dispatch($requestedPath) {
        if (isset($this->routes[$requestedPath])) {
            call_user_func($this->routes[$requestedPath]);
        } else {
            http_response_code(404);
            echo "Erreur 404 : Page non trouvée.";
        }
    }
}
```

Le fichier **Router.php** contient la classe **Router**, qui sert à gérer les routes de l'application. Une route détermine la manière dont une requête HTTP est traitée par l'application. Ce fichier gère les chemins des URL et les actions (callbacks) qui doivent être exécutées lorsque ces chemins sont demandés.

## 1. Propriétés de la classe

- **\$routes** :
  - C'est un tableau associatif qui contient les routes de l'application. La clé du tableau est le chemin de la route (par exemple, `/home` ou `/animals`), et la valeur est un **callback** (une fonction ou une méthode) qui sera exécutée lorsque cette route est demandée.
  - Ce tableau est initialisé vide, mais il est rempli via la méthode **add()** qui permet d'ajouter des routes à l'application.

## 2. Méthode `add($path, $callback)`

- **Rôle** : Cette méthode permet d'ajouter de nouvelles routes à l'application en associant un chemin d'URL à une action spécifique (un callback).
- **Paramètres** :
  - **\$path** : C'est le chemin de la route. Par exemple, `/animals` ou `/contact`.
  - **\$callback** : C'est la fonction ou la méthode qui sera appelée lorsqu'une requête pour cette route sera reçue. Le callback peut être une fonction anonyme ou une méthode d'un contrôleur.
- **Fonctionnement** :

- Lorsqu'une route est ajoutée, elle est enregistrée dans le tableau associatif **\$routes** avec le chemin **\$path** comme clé et le **callback** comme valeur.
- Par exemple, si tu ajoutes une route avec `add( '/animals' , 'showAnimals' )`, lorsque l'URL `/animals` est demandée, la méthode `showAnimals` sera appelée.

### 3. Méthode **dispatch(\$requestedPath)**

- **Rôle** : Cette méthode est responsable du traitement d'une requête HTTP et de l'exécution du callback associé à la route demandée.
- **Paramètres** :
  - **\$requestedPath** : C'est le chemin de la route demandée par l'utilisateur. Par exemple, si l'URL de la page est `http://example.com/animals`, alors le chemin demandé sera `/animals`.
- **Fonctionnement** :
  - La méthode vérifie si la route demandée (c'est-à-dire, le chemin **\$requestedPath**) existe dans le tableau **\$routes**.
  - Si la route existe, elle exécute le **callback** associé à cette route en utilisant `call_user_func()`.
  - Si la route n'existe pas dans le tableau, un code d'état HTTP **404** est renvoyé (erreur de page non trouvée) et un message d'erreur est affiché à l'utilisateur : "**Erreur 404 : Page non trouvée.**".

### 4. Résumé de la classe **Router**

- La classe **Router** permet de gérer la correspondance entre les routes demandées par l'utilisateur et les actions qui doivent être effectuées. Elle se compose de deux principales méthodes :
  - **add(\$path, \$callback)** pour enregistrer une nouvelle route et associer un chemin à une action.
  - **dispatch(\$requestedPath)** pour traiter une requête et exécuter l'action correspondante à la route demandée.
- Le **Router** offre une manière simple et flexible de gérer les requêtes HTTP et d'exécuter différentes actions en fonction du chemin de l'URL.
- **Intégration avec MongoDB** :
  1. Introduction à MongoDB dans le projet :

MongoDB est utilisé dans ce projet pour gérer les données qui ne nécessitent pas une structure relationnelle classique. Plus précisément, MongoDB est utilisé pour enregistrer le nombre de vues des animaux dans le zoo. Cela permet de suivre facilement l'interaction des visiteurs avec les animaux sans alourdir la base de données MySQL.

## 2. Connexion à MongoDB :

La connexion à MongoDB est gérée via la classe **MongoDBConnection**. Cette classe se connecte à la base de données **zoo** en utilisant l'URL de connexion de MongoDB et fournit une méthode pour accéder à une collection spécifique, comme **animal\_views**.

### Exemple de code pour la connexion :

```
*****  
$this->client = new MongoDB\Client("mongodb://localhost:27017");  
$this->db = $this->client->zoo; // Base de données 'zoo'
```

## 3. Structure des données dans MongoDB :

Les données des vues des animaux sont stockées dans une collection appelée **animal\_views**. Chaque document dans cette collection contient deux champs :

- **animal\_id** : L'identifiant unique de l'animal.
- **views\_count** : Le nombre de fois que cet animal a été consulté.

## 4. Gestion des vues des animaux :

À chaque fois qu'un utilisateur consulte la page d'un animal, nous mettons à jour son nombre de vues dans la collection MongoDB. Si un document pour cet animal existe déjà, on incrémente le compteur. Sinon, un nouveau document est créé avec un compteur initialisé à 1.

Exemple de code pour mettre à jour les vues :

```
$existingView = $collection->findOne(['animal_id' => (int)$id]);  
  
if ($existingView) {  
    // Si le document existe, mettre à jour le compteur  
    $collection->updateOne(  
        ['animal_id' => (int)$id],  
        ['$inc' => ['views_count' => 1]]  
    );  
} else {  
    // Si le document n'existe pas, créer un nouveau document  
    $collection->insertOne(  
        ['animal_id' => (int)$id,  
         'views_count' => 1  
    );
```

## 5. Avantages de MongoDB :

MongoDB est particulièrement adapté pour ce type de données :

- **Rapidité** : Les opérations de lecture et d'écriture sont rapides, ce qui est important pour le comptage des vues.
- **Flexibilité** : La structure de MongoDB est flexible, ce qui permet de stocker facilement des données sans avoir à respecter un schéma rigide.
- **Scalabilité** : Si le nombre d'animaux ou de vues augmente, MongoDB peut facilement évoluer pour gérer de grandes quantités de données.

## 6. Limitations de MongoDB :

Bien que MongoDB soit très performant pour des opérations simples et à grande échelle, il peut présenter des limitations dans la gestion de transactions complexes ou de relations entre les données. Cependant, pour notre cas d'usage (comptage de vues), ces limitations ne posent pas de problème.

## 7. Conclusion

L'intégration de MongoDB dans ce projet permet de gérer efficacement les données de consultations des animaux, avec une approche simple, rapide et évolutive. Cette solution complète bien l'utilisation de MySQL pour les données relationnelles et est parfaitement adaptée à notre besoin de suivi des vues.

## Design

- **Introduction à Figma** (screenshot des maquettes en annexe page 75) :
- Figma est un outil de conception d'interface utilisateur basé sur le cloud, largement utilisé pour la création de maquettes interactives et collaboratives. Il permet de concevoir des interfaces utilisateur (UI) modernes, de créer des prototypes interactifs, et de collaborer en temps réel avec les équipes de développement et de design. Dans le cadre de ce projet, Figma a été utilisé pour réaliser l'ensemble des maquettes visuelles du site, permettant de donner vie à la structure et à l'esthétique de l'application web.

- **Description de la Maquette :**
- La maquette réalisée dans Figma a été conçue pour offrir une interface claire et intuitive pour les utilisateurs tout en facilitant la navigation entre les différentes sections du site. Voici les principaux éléments de la maquette :
- **Navbar (Barre de navigation)** : La navbar est présente sur toutes les pages du site, offrant un accès rapide et facile aux différentes sections de l'application. Elle permet de naviguer entre :
  - **Accueil** : Page d'introduction présentant le zoo et ses valeurs écologiques.
  - **Animaux** : Affichage des animaux du zoo, leurs informations détaillées et leurs habitats.
  - **Habitats** : Détail des différents habitats du zoo et les animaux qui y résident.
  - **Services** : Informations sur les services offerts par le zoo (restauration, visites guidées, etc.).
  - **Avis** : Section permettant aux visiteurs de laisser des commentaires sur leur expérience au zoo.
  - **Horaires** : Informations sur les horaires d'ouverture du zoo.
- **Présentation du Zoo** : La page d'accueil présente le zoo, ses valeurs écologiques, et un aperçu visuel de ses différents aspects. Des images d'animaux et de paysages du zoo sont utilisées pour capturer l'attention des visiteurs.
- **Images des Animaux et Habitats** : Les pages des animaux et des habitats intègrent des images en haute qualité qui permettent aux visiteurs d'explorer visuellement les différentes espèces et leur environnement naturel.
- **Section Services** : Cette partie de la maquette met en avant les services offerts par le zoo (par exemple : visites guidées, restauration). Chaque service est représenté avec une petite description et une icône pour renforcer la lisibilité.
- **Avis des Visiteurs** : Les avis sont présentés sous forme de commentaires laissés par les visiteurs, affichant des informations comme le nom de l'auteur et la date de soumission. Un système de validation est prévu pour les employés du zoo.
- **Horaires du Zoo** : La maquette comprend également une section dédiée aux horaires d'ouverture du zoo, avec une présentation claire et facile à comprendre.
- **Footer (Pied de page)** : Le footer est présent sur chaque page et inclut :
  - **Mentions légales** : Lien vers les informations légales du site.
  - **Politique de confidentialité** : Lien vers la politique de confidentialité du zoo.
  - **Connexion à la partie administration** : Un lien permettant l'accès à la partie administrateur du site pour la gestion des animaux, des services, des avis, etc.

- **Conclusion**
- La maquette réalisée dans Figma permet de visualiser de manière fluide et cohérente l'interface du site, en mettant l'accent sur l'accessibilité, la navigation intuitive, et une expérience utilisateur agréable. Elle sert de référence pour l'intégration du design et du contenu dans l'application web, facilitant la collaboration entre les équipes de développement et de design.
- **Choix de design et adoption de Bootstrap :**
  1. **Introduction au choix de design :**
    - Dans le cadre de la création de notre site pour le zoo Arcadia, l'objectif principal était de proposer une interface claire, intuitive et responsive, capable de s'adapter à tous les types d'appareils (mobiles, tablettes, ordinateurs). Un design efficace est essentiel pour offrir une expérience utilisateur optimale, permettant aux visiteurs de naviguer facilement entre les différentes sections du site, découvrir les animaux ou encore consulter les horaires. La solution adoptée pour la création du design a été le framework Bootstrap, un outil flexible et puissant permettant de répondre à ces besoins.
  2. **Raisons du choix de Bootstrap :**
    - **Présentation de Bootstrap :**
    - Bootstrap est un framework CSS open-source largement utilisé dans le développement web. Il propose une collection de composants prêts à l'emploi, allant des boutons, formulaires, barres de navigation, aux systèmes de grilles pour structurer le contenu de manière fluide et responsive. Ce framework offre aussi des options avancées, comme les modales et les carrousels, pour intégrer facilement des éléments interactifs.
    - **Pourquoi avoir choisi Bootstrap ? :**
    - **Design responsive et mobile-first :** Étant donné que de nombreux utilisateurs accèdent aux sites via leurs téléphones mobiles, il était crucial que le site soit entièrement responsive. Bootstrap, avec son approche mobile-first, facilite cette tâche en permettant d'adapter l'affichage du site à toutes les tailles d'écran.
    - **Gain de temps et efficacité :** Bootstrap permet de gagner énormément de temps dans le développement en offrant des composants CSS et JavaScript prédéfinis. Les utilisateurs peuvent bénéficier immédiatement de styles modernes, tout en économisant des heures de codage pour chaque élément d'interface.
    - **Compatibilité avec PHP et JavaScript :** Le projet utilise PHP pour la logique côté serveur et JavaScript pour l'interactivité côté client. Bootstrap s'intègre facilement à ces technologies,

sans nécessiter de configuration complexe. De plus, il est possible de personnaliser le design et d'ajouter des éléments interactifs grâce aux composants JavaScript fournis.

- **Facilité de personnalisation :** Bien que Bootstrap fournit un design standard, il permet aussi une personnalisation poussée. Nous avons pu ajuster des éléments de style tels que les couleurs, les tailles de texte, et utiliser Flexbox pour améliorer la mise en page de certaines sections.
- **Intégration de Bootstrap dans le projet :**
- Bootstrap a été intégré au projet via un CDN (Content Delivery Network). Cette approche a simplifié le processus d'intégration, permettant de charger rapidement les fichiers nécessaires sans avoir à les héberger localement. L'intégration de Bootstrap a été fluide et n'a posé aucune difficulté majeure, grâce à la documentation complète et claire fournie par le framework.
- **Personnalisation de Bootstrap :**
- Afin d'adapter Bootstrap à l'identité visuelle du zoo Arcadia, certaines personnalisations ont été effectuées. Cela comprenait :
  - **Modification des couleurs et des tailles :** Nous avons ajusté les couleurs des boutons, des liens et des titres pour les aligner avec le thème du zoo, tout en conservant l'accessibilité et la lisibilité du site.
  - **Utilisation de Flexbox :** Certaines sections ont été conçues avec Flexbox pour garantir une disposition flexible et responsive, comme la mise en page des informations sur les animaux ou des horaires d'ouverture.
  - **Réajustement des espacements et des tailles de police :** Pour rendre le site plus aéré et agréable à lire, des ajustements de taille et d'espacement ont été faits, tout en restant fidèle à la structure générale de Bootstrap.
- **Exemples d'utilisation des composants Bootstrap :**
- **Barre de navigation (Navbar) :**

```
<nav class="navbar navbar-expand-lg navbar-dark bg-success sticky-top">
  <div class="container-sm">
    <a class="navbar-brand" href="/home">Arcadia
    </a>

    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link" aria-current="page" href="/home">Accueil</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/habitats">Habitats</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/services">Services</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/notice">Avis</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/contact">Contact</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```



- La barre de navigation permet aux utilisateurs de naviguer facilement entre les différentes pages du site (Accueil, Animaux, Réservations, etc.). Grâce aux classes Bootstrap, elle est entièrement responsive et se réorganise sur les petits écrans.
- **Cartes Bootstrap:**

```
<?php include 'template/clientHeader.php'; ?>
<?php include 'template/clientNavbar.php'; ?>

<div class="container mt-5">
  <h1 class="text-center my-4">Nos Habitats</h1>
  <div class="row">
    <?php foreach ($habitats as $habitat): ?>
    <div class="col-md-4 mb-4">
      <div class="card h-100">
         htmlspecialchars($habitat['name']) ?>" style="height: 200px; object-fit: cover;">
        <div class="card-body">
          <h2 class="card-title">?=> htmlspecialchars($habitat['name']) ?></h2>
          <p class="card-text">?=> htmlspecialchars($habitat['description']) ?></p>

          <!-- Liste des animaux -->
          <?php if (!empty($habitat['animals'])): ?>
            <ul class="list-group list-group-flush">
              <?php foreach ($habitat['animals'] as $animal): ?>
                <li class="list-group-item">
                  <a class="animale" href="/animal?id=?=> htmlspecialchars($animal['id']) ?>"><strong>?=> htmlspecialchars($animal['name']) ?></strong> (<?=> htmlspecialchars($animal['breed']) ?>)
                </li>
              <?php endforeach; ?>
            </ul>
          <?php else: ?>
            <p class="text-muted mt-3">Aucun animal dans cet habitat.</p>
          <?php endif; ?>
        </div>
      </div>
    <?php endforeach; ?>
  </div>
</?php>
```



**Savane**

La savane d'Arcadia est un vaste espace ensoleillé, parsemé d'herbes hautes et d'acacias, où les visiteurs peuvent observer des animaux majestueux évoluer dans leur environnement naturel. Ce paysage ouvert, ponctué de points d'eau, offre un habitat idéal pour les espèces qui cohabitent ici.

[Alex \(Lion\)](#)  
[Marty \(Zèbre\)](#)



**Jungle**

La jungle d'Arcadia est une forêt tropicale dense, riche en biodiversité, où la lumière filtre à travers un feuillage épais. Les sons des oiseaux exotiques et des cris des singes créent une atmosphère vibrante. Cet habitat humide et chaud abrite de nombreuses espèces fascinantes.

[Mister \(Toucan\)](#)

- Les cartes Bootstrap ont été utilisées pour afficher les informations sur chaque animal du zoo de manière claire et structurée.
- **Formulaire d'avis:**

```

<form id="review-form" method="POST" action="/submit-review" class="mt-4">
  <div class="mb-3">
    <label for="pseudo" class="form-label">Pseudo</label>
    <input type="text" class="form-control" id="pseudo" name="pseudo" required placeholder="Votre pseudo">
  </div>

  <div class="mb-3">
    <label class="form-label">Notation</label>
    <div id="rating" class="d-flex align-items-center">
      <input type="radio" id="star5" name="rating" value="5" required>
      <label for="star5" class="star">★</label>

      <input type="radio" id="star4" name="rating" value="4">
      <label for="star4" class="star">★</label>

      <input type="radio" id="star3" name="rating" value="3">
      <label for="star3" class="star">★</label>

      <input type="radio" id="star2" name="rating" value="2">
      <label for="star2" class="star">★</label>

      <input type="radio" id="star1" name="rating" value="1">
      <label for="star1" class="star">★</label>
    </div>
  </div>

  <div class="mb-3">
    <label for="comment" class="form-label">Commentaire</label>
    <textarea class="form-control" id="comment" name="comment" rows="5" required placeholder="Écrivez votre commentaire ici..."/>
  </div>

  <button type="submit" class="btn btn-success">Soumettre l'avis</button>
</form>
</div>

```

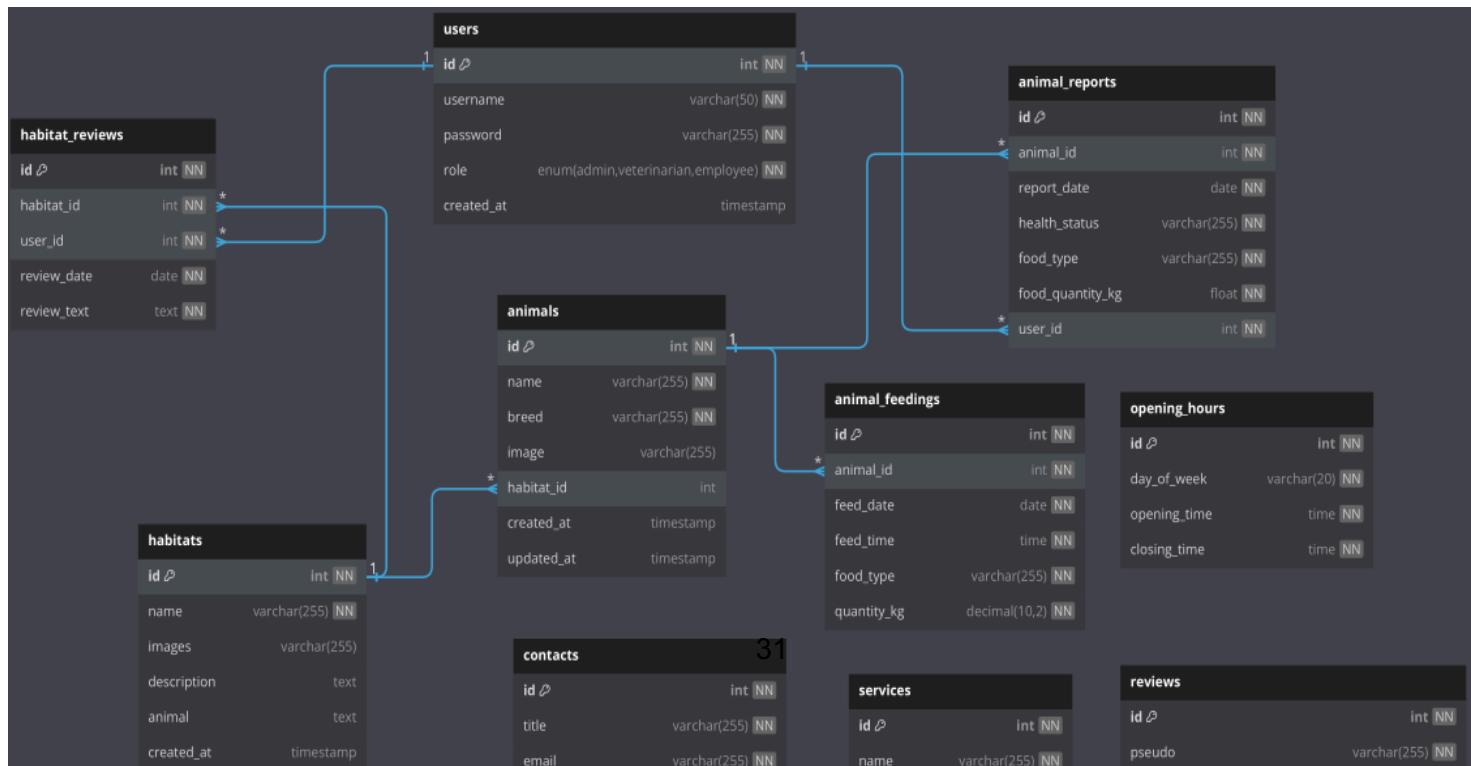
- Le site permet aux utilisateurs de laisser un commentaire de leurs expériences. Les formulaires ont été réalisés avec les composants Bootstrap pour garantir une présentation soignée et

une bonne ergonomie et avec la personnalisation, un système de notation étoile a été ajouté.

- **Les défis rencontrés :**
- L'intégration de Bootstrap s'est passée sans encombre grâce à sa documentation détaillée. Cependant, lors de la personnalisation des styles, certains conflits sont apparus entre le CSS de Bootstrap et nos styles personnalisés. Par exemple, certains éléments comme les boutons ont présenté des problèmes de mise en forme lorsque les classes Bootstrap entraient en conflit avec des styles personnalisés. Heureusement, ces problèmes ont été rapidement résolus grâce à une gestion appropriée des sélecteurs CSS et à l'utilisation de classes spécifiques pour surcharger les styles.
- Conclusion :
- L'utilisation de Bootstrap a été un choix stratégique qui a permis de créer un design moderne et réactif pour le site du zoo Arcadia. Sa flexibilité, sa facilité d'intégration et ses composants prêts à l'emploi ont accéléré le processus de développement. Malgré quelques ajustements nécessaires pour personnaliser le design, Bootstrap a permis de répondre à tous les besoins du projet en termes de fonctionnalité et d'esthétique.

## Base de données

- **Diagramme MLD (dbdiagram.io) :**



- **Structure des tables MySQL :**

- 1. Introduction à la base de données :**

- La base de données MySQL est au cœur de notre projet. Elle stocke toutes les informations essentielles liées aux différentes fonctionnalités du site du zoo Arcadia, telles que les informations sur les animaux, les réservations de billets, les utilisateurs, etc. Pour garantir une gestion optimale des données et faciliter leur récupération, plusieurs tables ont été créées, chacune avec une structure spécifique.

- 2. Tables de la base de données MySQL et leurs rôles :**

- Table animals :

Nom de la colonne	Type	Description
id	int	Identifiant unique de chaque animal. Il s'agit de la clé primaire de la table, utilisée pour distinguer chaque enregistrement d'animal. Ce champ est un entier et ne peut pas être nul.
name	varchar(255)	Nom de l'animal (par exemple, "Alex", "Marty", etc.). Ce champ est obligatoire pour chaque animal, car il permet de l'identifier clairement.
breed	varchar(255)	Race ou espèce de l'animal (par exemple, "Lion", "Crocodile", etc.). Ce champ permet de préciser la catégorie ou sous-catégorie de l'animal et est également obligatoire.
image	varchar(255)	URL ou chemin d'accès à une image représentant l'animal. Ce champ est facultatif (il peut être NULL si aucune image n'est fournie), mais il permet d'afficher une photo de l'animal sur le site.
habitat_id	int	Identifiant du lieu ou habitat où se trouve l'animal dans le zoo (lié à la table habitats si elle existe). Ce champ est facultatif et peut être utilisé pour établir des relations avec d'autres tables décrivant les différents habitats des animaux dans le zoo.

created_at	timestamp	Date et heure de création de l'enregistrement de l'animal. Cette colonne est automatiquement remplie avec la date et l'heure actuelles lors de l'ajout de l'animal dans la base de données.
updated_at	timestamp	Date et heure de la dernière mise à jour de l'enregistrement de l'animal. Ce champ est mis à jour automatiquement chaque fois qu'une modification est effectuée sur l'animal.

### Rôle de la table dans le projet

La table `animals` joue un rôle central dans le projet en permettant de stocker et de gérer les informations sur les animaux du zoo Arcadia. Elle permet de :

- **Affichage des informations sur les animaux** : Toutes les informations concernant les animaux sont récupérées et affichées sur le site, y compris leur nom, leur race, et éventuellement leur image. Cela permet aux visiteurs du zoo de découvrir les animaux présents sur place.
- **Gestion de l'inventaire des animaux** : Cette table est utilisée pour ajouter, mettre à jour ou supprimer des enregistrements concernant les animaux du zoo. Grâce à l'attribut `updated_at`, il est facile de suivre l'historique des modifications.
- **Relation avec d'autres entités** : Le champ `habitat_id` permet de lier chaque animal à son habitat spécifique, ce qui facilite la gestion de la disposition des animaux dans le zoo. Cela peut également être utile pour générer des informations complémentaires sur l'animal, telles que son environnement naturel.
  - Table `animal_feedings` :

Nom de la colonne	Type	Description
<code>id</code>	int	Identifiant unique pour chaque enregistrement de repas. Il s'agit de la clé primaire de la table. Chaque ligne dans la table <code>animal_feedings</code> aura un id unique pour le repas enregistré.
<code>animal_id</code>	int	Référence à l'identifiant de l'animal dans la table <code>animals</code> . Cette colonne permet de lier chaque repas à un animal

		spécifique. C'est une clé étrangère qui assure la relation avec la table animals.
feed_date	date	Date du repas. Ce champ indique quand le repas a été donné à l'animal (par exemple, "2025-01-28").
feed_time	time	Heure du repas. Ce champ spécifie à quelle heure l'animal a été nourri (par exemple, "08:30:00").
food_type	varchar(255)	Type de nourriture donnée à l'animal (par exemple, "Viande", "Fruits", "Légumes", etc.). Ce champ permet de spécifier le régime alimentaire de l'animal.
quantity_kg	decimal(10,2)	Quantité de nourriture donnée, en kilogrammes (par exemple, "5.50" pour 5,5 kg). Ce champ enregistre la quantité exacte de nourriture donnée lors du repas.

### Rôle de la table dans le projet

La table `animal_feedings` joue un rôle crucial dans la gestion des repas des animaux du zoo. Elle permet de :

- **Suivre l'alimentation des animaux** : Chaque repas est enregistré avec la quantité et le type de nourriture donnée, ainsi que la date et l'heure. Cela permet de suivre les habitudes alimentaires des animaux, de s'assurer qu'ils reçoivent la quantité correcte de nourriture et que leur alimentation est variée.
- **Assurer une gestion efficace de l'alimentation** : Grâce à cette table, les gestionnaires du zoo peuvent analyser les habitudes alimentaires des animaux et ajuster leur régime en fonction des besoins nutritionnels spécifiques. Cela peut également être utile pour les vétérinaires et les soignants qui suivent l'état de santé des animaux.
- **Rapports et audits** : En cas de besoin (par exemple, pour des audits ou des contrôles vétérinaires), la table permet de générer des rapports sur l'alimentation des animaux, en donnant un aperçu détaillé des repas qui ont été donnés à chaque animal sur une période donnée.
  - Table `animal_reports` :

Nom de la colonne	Type	Description
id	int	Identifiant unique du rapport. Il

		s'agit de la clé primaire de la table, permettant d'identifier chaque rapport de manière unique.
animal_id	int	Référence à l'animal concerné, avec un identifiant qui fait le lien avec la table <code>animals</code> . Cette colonne est une clé étrangère et permet de lier chaque rapport à un animal précis.
report_date	date	Date à laquelle le rapport a été rédigé. Ce champ indique quand le rapport a été effectué pour l'animal, ce qui permet de suivre l'évolution de l'animal au fil du temps.
health_status	varchar(255)	État de santé de l'animal à la date du rapport. Ce champ permet de décrire brièvement l'état de santé de l'animal (par exemple, "En bonne santé", "Convalescence", "Malade", etc.).
food_type	varchar(255)	Type de nourriture que l'animal reçoit, qui peut être utilisé pour vérifier si son alimentation est adaptée à son état de santé ou à ses besoins spécifiques.
food_quantity_kg	float	Quantité de nourriture donnée à l'animal en kilogrammes, qui permet de suivre les quantités spécifiques de nourriture administrées.
user_id	int	Identifiant de l'utilisateur (personnel du zoo) qui a rédigé le rapport. Cette colonne fait référence à la table <code>users</code> (ou une table similaire) pour savoir quel employé a enregistré ce rapport.

#### Rôle de la table dans le projet

La table `animal_reports` a un rôle clé dans la gestion de la santé et de l'alimentation des animaux du zoo Arcadia. Elle permet de :

- **Suivre l'état de santé des animaux** : En enregistrant des rapports détaillant l'état de santé de chaque animal, la table permet aux vétérinaires et autres soignants de suivre l'évolution de la condition physique des animaux au fil du

temps. Cela facilite la gestion préventive des soins et permet de documenter l'historique médical des animaux.

- **Surveiller l'alimentation des animaux :** En enregistrant des informations sur la quantité et le type de nourriture donnée, cette table permet de garantir que les animaux reçoivent une alimentation adéquate, en fonction de leurs besoins spécifiques. Par exemple, un animal malade peut avoir besoin d'un régime spécial, et la table permet de le suivre précisément.
- **Identifier les soins nécessaires :** En lien avec les rapports sur l'état de santé, la table permet aux soignants de déterminer si des ajustements sont nécessaires dans l'alimentation ou dans les traitements médicaux de l'animal.
- **Responsabilité et traçabilité :** Grâce à la présence de la colonne `user_id`, il est possible de savoir quel membre du personnel a rédigé le rapport, assurant ainsi la traçabilité des interventions et des suivis effectués.

- Table users :

Nom de la colonne	Type	Description
id	int	Identifiant unique pour chaque utilisateur. Il s'agit de la clé primaire de la table, permettant d'identifier de manière unique chaque utilisateur dans le système.
username	varchar(50)	Nom d'utilisateur (login) utilisé pour se connecter au système. Ce champ est obligatoire et doit être unique pour chaque utilisateur.
password	varchar(255)	Mot de passe de l'utilisateur, stocké de manière sécurisée (par exemple, en utilisant un algorithme de hachage comme bcrypt). Ce champ est obligatoire pour garantir la sécurité des comptes.
role	enum('admin', 'veterinarian', 'employee')	Rôle de l'utilisateur dans le système. Cela permet de définir les permissions et les accès dans le site : un <code>admin</code> a un accès complet, un <code>veterinarian</code> peut consulter et ajouter des rapports sur les animaux, et un <code>employee</code> peut avoir des accès limités.
created_at	timestamp	Date et heure de création du compte utilisateur. Ce champ est automatiquement rempli avec la date et l'heure actuelles lors de la création du

		compte. Il permet de suivre l'historique des comptes créés.
--	--	---

## Rôle de la table dans le projet

La table `users` est centrale dans la gestion des accès et des permissions au sein du système du zoo. Elle permet de :

- **Gérer l'authentification des utilisateurs** : Le champ `username` permet à chaque utilisateur de se connecter, et le champ `password` assure la sécurité de l'authentification.
- **Attribuer des rôles et des permissions** : Grâce à la colonne `role`, chaque utilisateur peut se voir attribuer un rôle spécifique (admin, vétérinaire, employé). Ce système de rôles est important pour restreindre ou accorder des droits d'accès spécifiques à certaines parties du site ou du système, garantissant ainsi un contrôle précis de qui peut faire quoi. Par exemple :
  - **Administrateur (admin)** : Accès complet à toutes les fonctionnalités du site, y compris la gestion des utilisateurs, des animaux, des rapports, etc.
  - **Vétérinaire (veterinarian)** : Accès aux informations liées à la santé des animaux, avec la possibilité de créer ou modifier des rapports de santé.
  - **Employé (employee)** : Accès plus limité, probablement pour des tâches de gestion quotidienne ou des interactions avec les visiteurs, mais sans accès aux fonctionnalités administratives ou médicales.
- **Suivre l'historique des utilisateurs** : Grâce à la colonne `created_at`, il est possible de savoir quand chaque compte a été créé, ce qui peut être utile pour les audits de sécurité ou la gestion des utilisateurs.
  - Table habitats :

Nom de la colonne	Type	Description
<code>id</code>	<code>int</code>	Identifiant unique de l'habitat. Il s'agit de la clé primaire de la table, permettant d'identifier chaque habitat de manière unique.
<code>name</code>	<code>varchar(255)</code>	Nom de l'habitat (par exemple, "Savane africaine", "Forêt tropicale", etc.). Ce champ est obligatoire et permet d'identifier facilement l'habitat.
<code>images</code>	<code>varchar(255)</code>	URL ou chemin d'accès à une image représentant l'habitat. Ce champ est facultatif et peut être laissé à <code>NULL</code> si aucune

		image n'est associée à l'habitat. Cependant, il permet d'ajouter un visuel pour chaque habitat afin de mieux l'illustrer dans le site.
description	text	Description de l'habitat, expliquant ses caractéristiques, son environnement, et les conditions qui s'y trouvent (par exemple, "Cet habitat imite la savane d'Afrique avec des herbes sèches et des arbres épars"). Ce champ permet d'apporter plus de détails sur chaque espace.
animal	text	Liste des animaux qui vivent dans cet habitat. Ce champ peut contenir une description des espèces présentes, ou une liste des noms des animaux (par exemple, "Lions, Gazelles, Zèbres").
created_at	timestamp	Date et heure de création de l'enregistrement de l'habitat. Ce champ est automatiquement rempli lors de la création de l'enregistrement et permet de suivre la date à laquelle l'habitat a été ajouté.
updated_at	timestamp	Date et heure de la dernière mise à jour de l'enregistrement de l'habitat. Ce champ est automatiquement mis à jour chaque fois qu'une modification est effectuée sur l'habitat.

## Rôle de la table dans le projet

La table **habitats** est essentielle pour la gestion des différents espaces du zoo Arcadia. Elle permet de :

- **Organiser les espaces du zoo** : Chaque habitat correspond à un espace spécifique où les animaux vivent, et la table permet de décrire ces espaces de manière structurée. Cela permet aux visiteurs du site de découvrir chaque zone du zoo avec des informations complètes.
- **Associer des animaux aux habitats** : Grâce à la colonne **animal**, cette table permet de savoir quels animaux sont associés à chaque habitat. Cela est utile pour la gestion des animaux, afin de suivre quels animaux se trouvent dans quel espace et de mieux organiser la disposition des animaux.

- **Fournir des informations détaillées aux visiteurs :** Les informations stockées dans cette table, telles que le nom de l'habitat, sa description et ses images, sont essentielles pour présenter les habitats aux visiteurs du site web du zoo. Cela permet d'améliorer l'expérience des visiteurs en leur donnant une idée précise des différents espaces du zoo.
  - Table habitat\_reviews :

Nom de la colonne	Type	Description
id	int	Identifiant unique pour chaque avis. Il s'agit de la clé primaire de la table, permettant d'identifier chaque avis de manière unique.
habitat_id	int	Référence à l'identifiant de l'habitat concerné, faisant le lien avec la table <code>habitats</code> . Cette colonne est une clé étrangère qui associe chaque avis à un habitat spécifique.
user_id	int	Identifiant de l'utilisateur ayant laissé l'avis. Ce champ fait référence à la table <code>users</code> et permet de savoir quel utilisateur a rédigé l'avis.
review_date	date	Date à laquelle l'avis a été laissé. Ce champ indique quand l'utilisateur a rédigé son commentaire sur l'habitat.
review_text	text	Contenu du commentaire ou de l'avis laissé par l'utilisateur. Ce champ permet d'exprimer une évaluation détaillée de l'habitat, que ce soit sur son apparence, son aménagement, son accessibilité, etc.

## Rôle de la table dans le projet

La table `habitat_reviews` est importante pour recueillir les retours des utilisateurs et améliorer l'expérience des visiteurs du zoo. Elle permet de :

- **Collecter des avis sur les habitats :** Les utilisateurs (qu'ils soient visiteurs ou membres du personnel) peuvent laisser des avis détaillés sur chaque habitat, contribuant ainsi à une évaluation continue des espaces du zoo. Cela peut inclure des commentaires sur la propreté, l'aménagement, le confort des animaux, etc.
- **Suivre la satisfaction des visiteurs :** Les avis peuvent être utilisés pour analyser la satisfaction des visiteurs et identifier des domaines où des

améliorations peuvent être apportées, par exemple en ajustant les aménagements ou en modifiant certains aspects des habitats.

- **Favoriser l'interaction avec les utilisateurs :** La possibilité pour les utilisateurs de laisser des avis permet d'encourager l'interaction avec le site, tout en offrant un moyen pour les visiteurs de partager leurs impressions. Ces avis peuvent également être utilisés à des fins de promotion ou d'amélioration de l'image du zoo.
- **Gestion des commentaires des employés :** En plus des visiteurs, les employés peuvent aussi laisser des avis, par exemple pour signaler un problème dans un habitat ou donner des suggestions d'amélioration concernant l'environnement des animaux.

- Table reviews :

Nom de la colonne	Type	Description
id	int	Identifiant unique de l'avis. Il s'agit de la clé primaire de la table, permettant d'identifier chaque avis de manière unique.
pseudo	varchar(255)	Pseudo de l'utilisateur ayant laissé l'avis. Ce champ est obligatoire et permet d'identifier l'utilisateur sans avoir besoin de connaître son identité réelle.
rating	int	Note donnée par l'utilisateur, généralement sur une échelle de 1 à 5 (ou de 1 à 10, selon la configuration). Ce champ est obligatoire et permet d'évaluer de manière quantitative l'expérience de l'utilisateur.
comment	text	Commentaire détaillant l'avis de l'utilisateur. Ce champ est obligatoire et permet de recueillir des informations qualitatives sur l'expérience de l'utilisateur.
status	enum('pending', 'approved', 'rejected')	Statut de l'avis, indiquant s'il est en attente d'approbation, approuvé ou rejeté. Par défaut, un nouvel avis est marqué comme "pending" (en attente). Ce champ permet de gérer le contrôle qualité des avis avant leur publication.
created_at	timestamp	Date et heure de création de l'avis. Ce champ est automatiquement rempli lors de la création de l'avis et

		permet de suivre l'historique des avis.
--	--	---

## Rôle de la table dans le projet

La table `reviews` joue un rôle clé dans la gestion des retours d'expérience des utilisateurs concernant les différents aspects du zoo Arcadia. Elle permet de :

- **Recueillir des avis et évaluations des utilisateurs** : Les visiteurs peuvent laisser des commentaires sur leur expérience au zoo, qu'il s'agisse de la qualité des services, des installations, des attractions, ou de tout autre aspect de leur visite. Cela aide à évaluer la satisfaction des visiteurs et à améliorer les services du zoo.
- **Gérer l'approbation des avis** : Grâce à la colonne `status`, cette table permet de suivre l'état des avis. Les avis peuvent être soumis à une révision avant d'être publiés. Les avis en attente (status "pending") peuvent être examinés par les administrateurs avant d'être approuvés ou rejetés.
- **Suivre la qualité des services** : En analysant les notes (`rating`) et les commentaires (`comment`), le zoo peut identifier les points forts et les domaines nécessitant des améliorations. Par exemple, un avis avec une faible note et un commentaire négatif pourrait signaler un problème à résoudre.

- Table contacts :

Nom de la colonne	Type	Description
id	int	Identifiant unique du message de contact. Il s'agit de la clé primaire de la table, permettant d'identifier chaque message de manière unique.
title	varchar(255)	Titre ou objet du message envoyé par l'utilisateur. Ce champ est obligatoire et permet d'avoir un aperçu du contenu du message sans l'ouvrir.
email	varchar(255)	Adresse e-mail de l'utilisateur qui a envoyé le message. Ce champ est obligatoire et permet de répondre à l'utilisateur.
message	text	Contenu du message de l'utilisateur. Ce champ est obligatoire et contient le texte du message, qu'il s'agisse d'une question, d'une plainte

		ou d'une suggestion.
created_ad	timestamp	Date et heure de création du message. Ce champ est automatiquement rempli lors de la soumission du message et permet de suivre quand chaque message a été envoyé.
status	enum('pending', 'replied')	Statut du message. Indique si le message est encore en attente de réponse ( <b>pending</b> ) ou si une réponse a été fournie ( <b>replied</b> ). Par défaut, les messages sont marqués comme "pending" jusqu'à ce qu'une réponse soit apportée.

### Rôle de la table dans le projet

La table **contacts** joue un rôle essentiel dans la gestion des demandes de contact des utilisateurs. Elle permet de :

- **Recueillir les messages des utilisateurs** : Les utilisateurs peuvent soumettre leurs demandes via un formulaire de contact sur le site. Les informations collectées dans cette table comprennent le titre du message, l'adresse e-mail de l'utilisateur et le contenu du message, ce qui permet au personnel du zoo de répondre de manière appropriée.
- **Suivre l'état des messages** : Grâce à la colonne **status**, cette table permet de suivre l'état de chaque message. Les messages peuvent être marqués comme "pending" lorsqu'ils n'ont pas encore reçu de réponse, et comme "replied" une fois que la réponse a été envoyée à l'utilisateur. Cela facilite la gestion des demandes et permet d'assurer que toutes les demandes sont traitées.
- **Assurer une gestion efficace des demandes** : En ayant une base de données bien structurée avec les informations sur chaque demande, il devient plus facile de répondre rapidement et de garder un suivi des messages reçus.
  - Table opening\_hours :

Nom de la colonne	Type	Description
id	int	Identifiant unique de l'entrée dans la table des horaires. Il s'agit de la clé primaire de la table, permettant d'identifier chaque enregistrement de manière unique.
day_of_week	varchar(20)	Nom du jour de la semaine (par exemple, "Lundi", "Mardi",

		etc.). Ce champ est obligatoire et permet d'associer chaque horaire à un jour spécifique.
opening_hours	time	Heure d'ouverture du zoo pour ce jour de la semaine. Ce champ est obligatoire et spécifie à quelle heure le zoo ouvre ses portes.
closing_hours	time	Heure de fermeture du zoo pour ce jour de la semaine. Ce champ est obligatoire et spécifie à quelle heure le zoo ferme ses portes.

### Rôle de la table dans le projet

La table `opening_hours` est fondamentale pour la gestion des horaires du zoo Arcadia. Elle permet de :

- **Indiquer les horaires d'ouverture et de fermeture** : En stockant les horaires d'ouverture pour chaque jour de la semaine, cette table permet de communiquer clairement aux visiteurs les heures pendant lesquelles le zoo est accessible.
- **Adapter les horaires selon les jours** : Cette table permet de gérer des horaires différents en fonction des jours de la semaine. Par exemple, les horaires peuvent être différents le week-end et pendant la semaine, ce qui est couramment pratiqué pour les établissements comme les zoos.
- **Faciliter l'affichage dynamique des horaires** : Les informations stockées dans cette table peuvent être utilisées sur le site web pour afficher dynamiquement les horaires d'ouverture du zoo, selon le jour actuel, ce qui améliore l'expérience utilisateur en offrant des informations actualisées.
  - Table services :

Nom de la colonne	Type	Description
id	int	Identifiant unique du service. Il s'agit de la clé primaire de la table, permettant d'identifier chaque service de manière unique.
name	varchar(255)	Nom du service. Ce champ est obligatoire et permet de donner un titre ou une désignation à chaque service, tel que "Visite guidée", "Atelier pour enfants", etc.
description	text	Description détaillée du service. Ce champ est utilisé

		pour fournir des informations supplémentaires sur le service, telles que son contenu, sa durée, ses modalités, etc.
image	varchar(255)	URL ou chemin vers une image représentant le service. Ce champ est optionnel et permet d'ajouter une image pour illustrer le service dans l'interface utilisateur.
created_at	timestamp	Date et heure de création du service. Ce champ est automatiquement rempli lors de l'ajout du service dans la base de données et permet de suivre la création des services.

## Rôle de la table dans le projet

La table **services** joue un rôle essentiel dans la gestion des services proposés par le zoo Arcadia. Elle permet de :

- **Lister les services disponibles** : Cette table permet d'enregistrer tous les services offerts par le zoo, qu'ils soient temporaires (par exemple, un événement spécial) ou permanents (comme les visites guidées, les espaces de restauration, etc.).
- **Fournir des informations détaillées sur chaque service** : La description permet de donner aux utilisateurs des détails sur ce qu'implique chaque service, ce qui est crucial pour l'expérience utilisateur, car cela aide les visiteurs à prendre des décisions éclairées sur les services qu'ils souhaitent utiliser.
- **Ajouter des images pour illustrer les services** : Le champ **image** permet d'associer une image à chaque service, ce qui est utile pour améliorer la présentation visuelle sur le site web, attirant ainsi l'attention des visiteurs sur les différents services proposés.
- **Suivre l'historique des services** : Le champ **created\_at** permet de connaître la date de création du service, facilitant ainsi la gestion des services dans le temps.
- **Intégration de MongoDB** :
  1. Introduction à MongoDB :
    - MongoDB est une base de données NoSQL orientée document, idéale pour gérer des données flexibles et non structurées. Dans ce projet, MongoDB a été intégré pour suivre les consultations par animal, en particulier le nombre de fois qu'un animal a été consulté, et pour afficher ces informations dans l'espace administrateur du zoo. Ce choix a été fait pour sa

facilité d'utilisation et sa performance dans la gestion de données non relationnelles.

## 2. Étapes de l'intégration :

- **Installation et configuration :**
- MongoDB a été installé localement sur la machine de développement, en utilisant le port par défaut **27017**. Une fois installé, la bibliothèque MongoDB a été intégrée au projet via **Composer**, ce qui permet une gestion simple des dépendances.
- **Connexion à MongoDB :**
- La connexion entre l'application PHP et MongoDB se fait via une classe dédiée **MongoDBConnection**, qui initialise la connexion à la base de données **zoo** et permet de récupérer les collections nécessaires pour l'application. Voici un extrait de code illustrant la connexion à MongoDB :

```
require '../vendor/autoload.php';

class MongoDBConnection {
    private $client;
    private $db;

    public function __construct() {
        // Connexion à MongoDB
        $this->client = new MongoDB\Client("mongodb://localhost:27017"); // Assurez-vous que le port et l'URL sont corrects
        $this->db = $this->client->zoo; // Base de données 'zoo'
    }

    public function getCollection($collectionName) {
        return $this->db->$collectionName; // Retourne la collection demandée
    }
}
```

- **Modélisation des données avec MongoDB :**
- Une seule collection MongoDB a été créée dans ce projet, appelée **animal\_views**. Cette collection permet de suivre le nombre de consultations de chaque animal. Chaque document dans la collection contient les informations suivantes :
- **animal\_id** : Identifiant unique de l'animal, lié à la base de données MySQL pour les informations détaillées.
- **views\_count** : Nombre de consultations de cet animal.
- **\_id** : Identifiant unique généré automatiquement par MongoDB pour chaque document.
- Voici un exemple de document dans la collection **animal\_views** :

```
{
    "_id": {
        "$oid": "676e994e80d44da7bc061ee2"
    },
    "animal_id": 2,
    "views_count": 4
},
{
    "_id": {
        "$oid": "676ea493876c312984021322"
    },
    "animal_id": 6,
    "views_count": 8
},|
```

- Chaque fois qu'un animal est consulté sur le site, le champ **views\_count** est mis à jour pour refléter le nombre de consultations. Si l'animal n'a pas encore de données dans la collection, un nouveau document est inséré.
- **Opérations MongoDB :**
- Voici les principales opérations réalisées pour gérer les consultations des animaux dans MongoDB.
- **Ajouter ou mettre à jour le compteur de consultations :**  
Lorsqu'un utilisateur consulte une page détaillant un animal, le compteur de consultations dans MongoDB est mis à jour ou créé si nécessaire.
- Exemple de code dans le contrôleur **AnimalController.php** :

```
public function showAnimalDetails() {
    $id = $_GET['id'] ?? null;

    if (!$id) {
        echo "Identifiant de l'animal manquant.";
        return;
    }

    $animal = $this->animalModel->getAnimalById($id);

    if (!$animal) {
        echo "Animal introuvable.";
        return;
    }

    // Ajouter ou mettre à jour le compteur de consultations dans MongoDB
    $mongo = new MongoDBConnection();
    $collection = $mongo->getCollection('animal_views');

    // Chercher si un document existe déjà pour cet animal
    $existingView = $collection->findOne(['animal_id' => (int)$id]);

    if ($existingView) {
        // Si le document existe, mettre à jour le compteur
        $collection->updateOne(
            ['animal_id' => (int)$id],
            ['$inc' => ['views_count' => 1]] // Incrémenter le compteur de vues
        );
    } else {
        // Si le document n'existe pas, créer un nouveau document
        $collection->insertOne([
            'animal_id' => (int)$id,
            'views_count' => 1 // Première consultation
        ]);
    }
}
```

- **Récupérer le nombre de consultations pour un animal spécifique :** Lors de la consultation des détails d'un animal, le nombre de consultations est récupéré via une méthode dans le modèle **Animal.php** :

```
public function getAnimalViews($animalId) {
    $viewData = $this->collection->findOne(['animal_id' => (int)$animalId]);

    return $viewData ? $viewData['views_count'] : 0;
}
```

- **Gestion des erreurs et des exceptions :**
- Les erreurs potentielles liées à MongoDB (par exemple, une connexion échouée ou une requête mal formulée) sont gérées à l'aide de blocs **try-catch**. Voici un exemple de gestion des erreurs de connexion à MongoDB :

```
try {
    $client = new MongoDB\Client("mongodb://localhost:27017"); // Connexion à MongoDB
    $database = $client->zoo; // Base de données 'zoo'
    $collection = $database->animal_views; // Collection 'animal_views'
} catch (MongoDB\Driver\Exception\Exception $e) {
    echo "Erreur de connexion à MongoDB : ", $e->getMessage();
}
```

- **Conclusion :**
- L'intégration de MongoDB dans ce projet permet de suivre efficacement les consultations des animaux. Chaque fois qu'un utilisateur consulte la fiche d'un animal, le compteur dans MongoDB est mis à jour en temps réel. Cette approche flexible et performante permet de gérer facilement les données de consultation sans alourdir la base de données relationnelle. Grâce à cette solution, l'affichage des données dans l'espace admin devient plus rapide et plus réactif.

## Développement backend

- **Fonctionnalités principales :**
- **Création de comptes vétérinaires et employés :**

- Dans le cadre de la gestion des utilisateurs du zoo, une des fonctionnalités principales de l'interface d'administration consiste à créer des comptes pour les **vétérinaires** et **employés**. Chaque utilisateur a un rôle spécifique, ce qui permet de gérer les droits d'accès et les actions qu'ils peuvent effectuer sur le site. Par exemple, un vétérinaire pourra consulter et ajouter des rapports sur la santé des animaux, tandis qu'un employé pourra gérer des informations relatives aux animaux, habitats et services. Cette fonctionnalité repose sur l'implémentation d'un modèle de gestion des utilisateurs en PHP avec une base de données MySQL. Nous allons passer en revue le processus de création de comptes pour ces utilisateurs, ainsi que l'architecture de la gestion de ces comptes.
- **Structure du modèle User.php :**
- Le modèle **User.php** gère les opérations liées aux utilisateurs dans la base de données. Il inclut des méthodes permettant de récupérer un utilisateur par son nom d'utilisateur ou son ID, mais aussi de créer un nouvel utilisateur avec un rôle spécifique. Voici comment cela fonctionne:

```
public function create($username, $password, $role) {
    try {
        // Valider le rôle
        $validRoles = ['veterinarian', 'employee', 'admin'];
        if (!in_array($role, $validRoles)) {
            throw new Exception("Rôle invalide.");
        }

        // Préparer la requête d'insertion
        $query = "INSERT INTO users (username, password, role) VALUES (:username, :password, :role)";
        $stmt = $this->conn->prepare($query);

        // Hacher le mot de passe
        $hashedPassword = password_hash($password, PASSWORD_BCRYPT);

        // Lier les paramètres
        $stmt->bindParam(':username', $username);
        $stmt->bindParam(':password', $hashedPassword);
        $stmt->bindParam(':role', $role);

        // Exécuter la requête
        return $stmt->execute();
    } catch (Exception $e) {
        // Log l'erreur (facultatif)
        error_log("Erreur lors de la création de l'utilisateur : " . $e->getMessage());
        return false;
    }
}
```

- **Validation du rôle** : Avant de créer un utilisateur, le rôle spécifié (vétérinaire, employé ou administrateur) est validé. Si le rôle n'est pas valide, une exception est levée.
- **Hachage du mot de passe** : Le mot de passe fourni par l'utilisateur est haché à l'aide de la fonction **password\_hash()**, ce qui permet de sécuriser les informations sensibles.

- **Requête SQL** : Une requête préparée est utilisée pour insérer le nouvel utilisateur dans la table `users`.
- **Récupération des utilisateurs** :
- Le modèle `User` propose également des méthodes pour récupérer les informations d'un utilisateur, soit par son nom d'utilisateur (`getUserByUsername()`), soit par son ID (`getUserById()`).

```

public static function getUserByUsername($username) {
    $db = Database::getInstance()->getConnection();
    $stmt = $db->prepare("SELECT * FROM users WHERE username = :username LIMIT 1");
    $stmt->bindParam(':username', $username, PDO::PARAM_STR);
    $stmt->execute();

    if ($stmt->rowCount() > 0) {
        return $stmt->fetch(PDO::FETCH_ASSOC);
    } else {
        return null; // Retourner null si l'utilisateur n'existe pas
    }
}

public function getUserId($id) {
    $sql = "SELECT id, role FROM users WHERE id = :id";
    $stmt = $this->conn->prepare($sql);
    $stmt->bindParam(':id', $id, PDO::PARAM_INT);
    $stmt->execute();
    return $stmt->fetch(PDO::FETCH_ASSOC);
}

```

- `getUserByUsername()` : Cette méthode permet de récupérer un utilisateur en fonction de son nom d'utilisateur. Elle est utilisée pour vérifier si un utilisateur existe déjà dans la base de données avant de créer un nouveau compte.
- `getUserById()` : Cette méthode permet de récupérer un utilisateur par son ID, avec un accès à des informations supplémentaires comme son rôle.
- **Contrôleur pour la création de comptes** :
- Dans le contrôleur `UserController.php`, voici comment l'administrateur pourrait appeler la méthode `createUser()` pour ajouter un vétérinaire ou un employé.

```

public function createUser() {
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        // Récupérer les données du formulaire
        $username = trim($_POST['username']);
        $password = trim($_POST['password']);
        $role = trim($_POST['role']); // Récupérer le rôle sélectionné

        // Valider les entrées
        if (empty($username) || empty($password) || empty($role)) {
            $errorMessage = "Tous les champs sont obligatoires.";
            require_once __DIR__ . '/../views/admin/create_user.php';
            return;
        }

        // Vérifier que le rôle est valide (par rapport à l'ENUM)
        $validRoles = ['veterinarian', 'employee'];
        if (!in_array($role, $validRoles)) {
            $errorMessage = "Rôle invalide. Veuillez choisir un rôle valide.";
            require_once __DIR__ . '/../views/admin/create_user.php';
            return;
        }

        // Créer un nouvel utilisateur
        $userCreated = $this->model->create($username, $password, $role);

        // Si l'utilisateur a été créé avec succès
        if ($userCreated) {
            $successMessage = "Utilisateur créé avec succès !";
        } else {
            $errorMessage = "Une erreur est survenue lors de la création de l'utilisateur.";
        }

        // Afficher à nouveau le formulaire avec le message de succès ou d'erreur
        require_once __DIR__ . '/../views/admin/create_user.php';
    } else {
        // Si ce n'est pas une requête POST, on affiche le formulaire de création d'utilisateur vide
        require_once __DIR__ . '/../views/admin/create_user.php';
    }
}

```

- Récupération des données** : Lorsque l'utilisateur soumet le formulaire, les données sont récupérées avec `$_POST` (nom d'utilisateur, mot de passe, et rôle).
- Validation des entrées** : Le code vérifie si tous les champs sont remplis. Si un champ est vide, un message d'erreur s'affiche. Le rôle sélectionné est ensuite vérifié pour s'assurer qu'il fait partie des rôles valides (vétérinaire ou employé).
- Création de l'utilisateur** : Si les données sont valides, la méthode `create()` du modèle `User.php` est appelée pour insérer l'utilisateur dans la base de données.

- **Affichage des messages** : Si la création de l'utilisateur réussit, un message de succès est affiché. Sinon, un message d'erreur est affiché.
- **Affichage du formulaire** : Le formulaire est réaffiché avec un message de succès ou d'erreur. Si la requête n'est pas une requête POST, le formulaire vide est affiché pour permettre à l'admin de créer un utilisateur.
- **Connexion d'un utilisateur** :
- La méthode **loginUser()** permet à un utilisateur de se connecter. Si les identifiants sont corrects, l'utilisateur est redirigé vers le tableau de bord correspondant à son rôle (admin, vétérinaire, ou employé) :

```
public function loginUser() {
    // Récupérer les données du formulaire
    $username = $_POST['username'];
    $password = $_POST['password'];

    // Vérifier si l'utilisateur existe dans la base de données
    $user = User::getUserByUsername($username);

    if ($user && password_verify($password, $user['password'])) {
        // L'utilisateur est authentifié, enregistrer la session
        $_SESSION['user_id'] = $user['id'];
        $_SESSION['username'] = $user['username'];
        $_SESSION['role'] = $user['role']; // Assurer que le rôle est dans la session

        // Rediriger l'admin vers la page du tableau de bord
        if ($_SESSION['role'] === 'admin') {
            header('Location: /admin-dashboard');
            exit();
        } elseif ($_SESSION['role'] === 'employee') {
            // Redirection vers une autre page selon le rôle (par exemple, employé)
            header('Location: /employee-dashboard');
            exit();
        } elseif ($_SESSION['role'] === 'veterinarian') {
            // Redirection vers une autre page selon le rôle (par exemple, vétérinaire)
            header('Location: /veto-dashboard');
            exit();
        }
    } else {
        // Si l'authentification échoue
        $errorMessage = "Identifiants incorrects.";
        require_once __DIR__ . '/../views/login.php';
    }
}
```

- **Récupération des données** : Les identifiants de connexion (nom d'utilisateur et mot de passe) sont récupérés via **\$\_POST**.
- **Vérification des identifiants** : Le contrôleur appelle **getUserByUsername()** pour vérifier si l'utilisateur existe. Ensuite, il

compare le mot de passe entré avec celui stocké dans la base de données via la fonction `password_verify()`.

- **Authentification réussie :**
- Si l'utilisateur est authentifié avec succès, ses informations (ID, nom d'utilisateur, rôle) sont stockées dans la session.
- L'utilisateur est ensuite redirigé vers la page appropriée en fonction de son rôle (admin, employé, vétérinaire).
- **Authentification échouée :** Si les identifiants sont incorrects, un message d'erreur est affiché et l'utilisateur est redirigé vers la page de login.
- **Modification des horaires du zoo :**
- Cette fonctionnalité permet à un administrateur de **modifier les horaires d'ouverture et de fermeture** du zoo pour chaque jour de la semaine. Le modèle `OpeningHours.php` gère la récupération et la mise à jour des horaires dans la base de données. Voici une explication détaillée des méthodes dans ce modèle.
- **Structure du modèle OpeningHours.php :**
- **Connexion à la base de données :**
- La classe `OpeningHours` utilise un objet `Database` pour obtenir une connexion à la base de données. Elle est donc prête à interagir avec la table `opening_hours`.

```
public function __construct() {
    $this->db = (new Database())->getConnection();
}
```

- **Récupérer tous les horaires :**
- La méthode `getAll()` permet de récupérer **tous les horaires** du zoo enregistrés dans la table `opening_hours`.

```
public function getAll() {
    $query = "SELECT * FROM opening_hours";
    $stmt = $this->db->query($query);
    return $stmt->fetchAll(PDO::FETCH_ASSOC);
}
```

- **Explication :** Elle effectue une requête `SELECT` pour obtenir toutes les lignes de la table `opening_hours` et retourne les résultats sous forme de tableau associatif.
- **Récupérer les horaires sous une forme spécifique :**

- La méthode **getingAll()** fait pratiquement la même chose que la précédente, mais elle formate les horaires de manière à ce que chaque jour de la semaine soit associé à une chaîne qui combine l'heure d'ouverture et de fermeture.

```
public function getingAll() {
    $sql = "SELECT * FROM opening_hours";
    $stmt = $this->db->prepare($sql);
    $stmt->execute();
    $results = $stmt->fetchALL(PDO::FETCH_ASSOC);

    // Retourne les horaires sous forme de tableau associatif
    $hours = [];
    foreach ($results as $row) {
        $hours[$row['day_of_week']] = $row['opening_time'] . ' - ' . $row['closing_time'];
    }
    return $hours;
}
```

- **Explication :** Elle transforme les résultats de la base de données en un tableau où chaque jour de la semaine (par exemple, "Monday", "Tuesday") est une clé et l'heure d'ouverture et de fermeture est une valeur sous la forme **HH:MM – HH:MM**.
- **Mise à jour des horaires :**
- La méthode **update()** permet de mettre à jour les horaires pour un jour donné dans la base de données. Elle prend en paramètre l'ID de la ligne à mettre à jour, l'heure d'ouverture et l'heure de fermeture.

```
public function update($id, $openingTime, $closingTime) {
    $query = "UPDATE opening_hours SET opening_time = :opening_time, closing_time = :closing_time WHERE id = :id";
    $stmt = $this->db->prepare($query);
    $stmt->bindParam(':id', $id);
    $stmt->bindParam(':opening_time', $openingTime);
    $stmt->bindParam(':closing_time', $closingTime);
    return $stmt->execute();
}
```

- **Explication :** Cette méthode utilise une requête **UPDATE** pour modifier les heures d'ouverture et de fermeture pour un jour spécifique, en fonction de l'ID du jour (qui doit être unique dans la table).
- **Structure du contrôleur OpeningHoursController.php :**
- **Initialisation du contrôleur :**
- Le contrôleur initialise le modèle **OpeningHours** dans son constructeur. Cela permet d'utiliser les méthodes du modèle pour manipuler les horaires dans la base de données.

```
public function __construct() {
    $this->model = new OpeningHours();
}
```

- **Affichage et mise à jour des horaires (Méthode `showHours`) :**
- La méthode `showHours` gère la logique d'affichage des horaires ainsi que la mise à jour de ces derniers.
- **Si la requête est de type POST** (lorsque l'administrateur soumet un formulaire pour mettre à jour les horaires) :
  - Elle parcourt chaque jour de la semaine et met à jour les horaires pour chaque jour en appelant la méthode `update()` du modèle.
  - Ensuite, elle définit un message de succès et récupère à nouveau les horaires mis à jour à partir du modèle.
  - Enfin, elle charge la vue `opening_hours.php` pour afficher les horaires mis à jour avec un message de succès.

```
public function showHours() {
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        foreach ($_POST['hours'] as $id => $times) {
            $this->model->update($id, $times['opening'], $times['closing']);
        }
        $successMessage = "Horaires mis à jour avec succès !";
        $hours = $this->model->getAll();
        require_once __DIR__ . '/../views/admin/opening_hours.php';
    } else {
        $hours = $this->model->getAll();
        require_once __DIR__ . '/../views/admin/opening_hours.php';
    }
}
```

- **Si la requête est de type GET** (lorsque l'administrateur consulte simplement les horaires sans modification) :
- Elle récupère tous les horaires actuels via la méthode `getAll()` du modèle.
- Puis elle affiche la vue avec les horaires actuels du zoo (sans message de succès).
- **Récupérer les horaires d'ouverture pour les visiteurs (Méthode `getPublicHours`) :**
- La méthode `getPublicHours` est utilisée pour récupérer les horaires d'ouverture du zoo sous une forme adaptée à l'affichage public.

```
public function getPublicHours() {
    $openingHoursModel = new OpeningHours();
    return $openingHoursModel->getingAll(); // Récupérer les horaires sans afficher la vue admin
}
```

- Elle utilise la méthode `getingAll()` du modèle `OpeningHours` pour récupérer les horaires sous forme de tableau associatif, où chaque jour est associé à ses horaires d'ouverture et de fermeture.
- **Mettre à jour les horaires (Méthode `updateHours`) :**
- La méthode `updateHours` permet de mettre à jour les horaires via un formulaire envoyé en **POST**. Elle fonctionne de manière similaire à `showHours` mais sans afficher immédiatement les horaires mis à jour.

```

public function updateHours() {
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        foreach ($_POST['hours'] as $id => $times) {
            $this->model->update($id, $times['opening'], $times['closing']);
        }
        header('Location: /showHours&success=true');
        exit();
    }
}

```

- Elle parcourt les horaires envoyés par le formulaire et appelle la méthode **update()** du modèle pour mettre à jour chaque horaire.
- Après la mise à jour, elle redirige l'utilisateur vers la page des horaires avec un paramètre de succès dans l'URL.
- **Gestion des mails avec Mailpit :**
- **Présentation de Mailpit :**
- Mailpit est un outil de test d'emails permettant de capturer et de visualiser les emails envoyés par une application dans un environnement local. Il permet aux développeurs de tester l'envoi d'emails sans que ces derniers soient réellement envoyés aux destinataires, ce qui est particulièrement utile lors du développement d'applications web avec des fonctionnalités liées à l'email. Mailpit dispose d'une interface web simple et intuitive pour visualiser facilement les emails capturés, inspecter leurs en-têtes et leur contenu. En utilisant Mailpit dans un environnement de développement local, il devient possible de tester l'envoi de mails dans des scénarios variés, tels que l'envoi de confirmations de formulaires ou de notifications, tout en évitant d'envoyer des emails en production ou de polluer les boîtes de réception des utilisateurs.
- **Utilisation de Mailpit dans le projet :**
- Dans ce projet web pour le zoo, Mailpit a été intégré pour gérer l'envoi et la réception des emails lors du remplissage des formulaires de contact par les visiteurs. Lorsque ces derniers soumettent un message via le formulaire, un email est généré et envoyé. L'employé du zoo pourra ensuite consulter ces messages et y répondre via l'interface Mailpit. L'outil a été installé manuellement en suivant les instructions fournies dans la documentation officielle sur GitHub. Pour l'envoi des emails, **PHPMailer** a été utilisé, et ce dernier a été installé via **Composer**, ce qui permet de gérer facilement les dépendances du projet et de garantir une gestion propre et évolutive des envois d'emails.
- **Installation et démarrage de Mailpit :**
- Mailpit a été installé manuellement en suivant les étapes de sa documentation officielle disponible sur GitHub (lien en annexe page ),**Depuis le terminal** : En utilisant une simple commande dans le terminal d'Ubuntu 24.04.  
Exemple : **mailpit**
- **Manuellement depuis son répertoire** : En se rendant dans le répertoire d'installation de Mailpit et en lançant l'application directement à partir de là.

- L'interface web de Mailpit est ensuite accessible via l'URL <http://localhost:8025>, où il est possible de visualiser les emails envoyés et capturés.
- **Configuration dans le projet :**
- Dans ce projet, l'intégration de Mailpit pour la gestion des mails se fait dans le contrôleur [ContactController.php](#). Lorsque le formulaire de contact est soumis, un email de confirmation est envoyé au visiteur en utilisant [PHPMailer](#), qui est configuré pour fonctionner avec Mailpit en tant que serveur SMTP. Le fichier [ContactController.php](#) gère à la fois l'envoi du message de contact et l'envoi d'une réponse à l'employé du zoo. Voici comment cela est configuré dans le code :
- **Envoi de l'email de confirmation**
- Lorsque le visiteur soumet un message via le formulaire de contact, un email de confirmation est envoyé à l'adresse du visiteur pour l'informer que son message a bien été reçu. La méthode [sendConfirmationEmail\(\)](#) est responsable de l'envoi de cet email.
- Le serveur SMTP est configuré comme suit dans le code :

```
private function sendConfirmationEmail($to, $title, $message) {
    $mail = new PHPMailer(true);

    try {
        // Configuration du serveur SMTP
        $mail->isSMTP();
        $mail->Host = 'localhost'; // Mailpit
        $mail->Port = 1025; // Port de Mailpit
        $mail->SMTPAuth = false; // Pas d'authentification pour Mailpit

        // Expéditeur et destinataire
        $mail->setFrom('noreply@yourdomain.com', 'Votre Entreprise');
        $mail->addAddress($to);

        // Contenu de l'email
        $mail->isHTML(true);
        $mail->Subject = "Confirmation de votre message : $title";
        $mail->Body = "<p>Merci pour votre message :</p><p><strong>$message</strong></p><p>Nous vous répondrons dans les plus brefs délais.</p>";

        $mail->send();
        return true;
    } catch (Exception $e) {
        // Vous pouvez logger l'erreur ici pour déboguer
        error_log("Erreur lors de l'envoi de l'email : " . $mail->ErrorInfo);
        return false;
    }
}
```

- **Envoi de la réponse à un visiteur**
- Lorsqu'un employé du zoo souhaite répondre à un message, la méthode [sendReply\(\)](#) permet d'envoyer la réponse. La configuration SMTP reste la même, avec Mailpit comme serveur local :

```

public function sendReply() {
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $id = (int)$_POST['id'];
        $email = trim($_POST['email']);
        $subject = trim($_POST['subject']);
        $message = trim($_POST['message']);

        if (!empty($email) && !empty($subject) && !empty($message)) {
            // Envoyer l'email avec PHPMailer
            $mail = new PHPMailer(true);
            try {
                $mail->isSMTP();
                $mail->Host = 'localhost';
                $mail->Port = 1025;
                $mail->SMTPAuth = false;

                $mail->setFrom('support@votreprojet.com', 'Support');
                $mail->addAddress($email);
                $mail->isHTML(true);
                $mail->Subject = $subject;
                $mail->Body = $message;

                $mail->send();

                header('Location: /manage-mails?success=envoi réussi');
                exit;
            } catch (Exception $e) {
                error_log("Erreur lors de l'envoi : " . $mail->ErrorInfo);
                header('Location: /reply?id=' . $id . '&error=mail_failed');
                exit;
            }
        } else {
            header('Location: /reply?id=' . $id . '&error=fields_required');
            exit;
        }
    }
}

```

- Résumé de la configuration :
- **Serveur SMTP** : `localhost`, utilisant Mailpit comme serveur SMTP local.
- **Port** : 1025 (port configuré par défaut pour Mailpit).
- **Authentification** : Aucun besoin d'authentification (`SMTPAuth = false`), car Mailpit ne nécessite pas d'authentification pour le test des emails locaux.
- **Expéditeur et destinataire** : L'email de confirmation ou de réponse est envoyé avec l'adresse de l'expéditeur configurée dans le code, et le destinataire est récupéré dynamiquement depuis le formulaire.
- **Visualisation des emails dans Mailpit** :
- Une fois que l'email est envoyé via le formulaire de contact, il est capturé par Mailpit et apparaît dans l'interface web, accessible via <http://localhost:8025>. Dans cette interface, il est possible de visualiser les emails envoyés, d'examiner leurs en-têtes, et de consulter le contenu complet des messages. Cela permet de s'assurer que l'email a été correctement formaté et envoyé avant de passer en production.
- **Conclusion** :
- Mailpit facilite grandement le développement et le test de la fonctionnalité d'envoi d'emails dans des applications locales, en offrant un moyen sécurisé et efficace de tester les mails sans risquer de perturber les utilisateurs finaux. Son utilisation dans ce projet pour le zoo permet de gérer les messages des

visiteurs de manière efficace tout en maintenant un environnement de développement propre et contrôlé.

- **Utilisation de Composer :**
- **Introduction à Composer :**
- **Composer** est un gestionnaire de dépendances pour PHP, utilisé pour gérer les bibliothèques et packages nécessaires au développement de ton application. Il facilite l'installation, la mise à jour et la gestion des dépendances de ton projet en automatisant ces tâches de manière centralisée. Dans ce projet, Composer a été utilisé pour installer des bibliothèques tierces, comme **PHPMailer**, et gérer les différentes dépendances nécessaires au bon fonctionnement de l'application.
- **Installation de Composer :**
- Avant de pouvoir utiliser Composer dans un projet PHP, il est nécessaire de l'installer sur ta machine. Voici les étapes pour installer Composer sur un système Ubuntu (version 24.04 dans ce cas) :
  - **Télécharger et installer Composer :**
  - Ouvre le terminal et exécute les commandes suivantes pour installer Composer globalement sur ta machine :

```
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
```
  - **Vérification de l'installation :**
  - Pour vérifier que Composer a été installé correctement, exécute la commande suivante dans le terminal :

```
composer --version
```
  - Cela devrait afficher la version installée de Composer, confirmant que l'installation a réussi.
  - **Utilisation de Composer dans le projet :**
  - Une fois Composer installé, tu peux l'utiliser pour gérer les dépendances de ton projet PHP. Dans ce projet, **Composer** a été utilisé pour installer plusieurs bibliothèques nécessaires, comme **PHPMailer** pour l'envoi d'emails, et **MongoDB** pour la gestion de la base de données NoSQL. Voici comment tu peux utiliser Composer pour gérer ces dépendances dans ton projet.
  - **Installation des dépendances :**
  - **PHPMailer** : Pour envoyer des emails depuis l'application, **PHPMailer** a été installé via Composer. La commande pour installer cette bibliothèque est la suivante :

```
composer require phpmailer/phpmailer
```
  - Cette commande ajoute **PHPMailer** à ton projet et met à jour le fichier **composer.json** pour enregistrer cette dépendance.
  - **MongoDB** : Pour interagir avec une base de données MongoDB, la bibliothèque officielle **mongodb/mongodb** a été installée. La commande suivante permet de l'ajouter au projet :

```
composer require mongodb/mongodb
```

- `composer require mongodb/mongodb`
- Cette bibliothèque permet de gérer les connexions et les opérations avec MongoDB dans ton projet, et elle est intégrée via le même processus de gestion de dépendances avec Composer. Composer télécharge alors ces bibliothèques et leurs dépendances dans le répertoire `vendor/`, et met à jour le fichier `composer.lock` pour garantir des installations cohérentes sur toutes les machines.
- **Autoloading des classes :**
- Une fois les dépendances installées via Composer, il est nécessaire d'inclure l'autoloader dans ton projet pour pouvoir utiliser ces bibliothèques. Dans ton fichier principal (par exemple `MongoDB.php` ou `ContactController.php`), tu dois inclure le fichier `vendor/autoload.php` :
- `require '../vendor/autoload.php';`
- Cela permet à Composer de charger automatiquement toutes les classes des bibliothèques installées, comme **PHPMailer** et **mongodb/mongodb**, sans avoir à les inclure manuellement dans chaque fichier.
- **Mise à jour des dépendances :**
- Si tu souhaites mettre à jour toutes les dépendances du projet vers leurs dernières versions compatibles avec les contraintes définies dans ton fichier `composer.json`, tu peux utiliser la commande suivante :
- `composer update`
- Cela permettra de mettre à jour toutes les bibliothèques et leurs dépendances à leur dernière version compatible.
- **Gestion des versions et des dépendances :**
- Composer permet également de spécifier des versions spécifiques des bibliothèques ou des intervalles de versions compatibles. Par exemple, dans le fichier `composer.json`, tu peux définir une version précise de PHPMailer :
- `"require": {`
- `"mongodb/mongodb": "^1.20",`
- `"phpmailer/phpmailer": "^6.9"`
- `}`
- Cela garantit que Composer installera une version compatible de **PHPMailer** avec les autres dépendances de ton projet.
- **Conclusion :**
- L'utilisation de Composer dans ce projet simplifie considérablement la gestion des dépendances et garantit une installation et une mise à jour faciles des bibliothèques nécessaires au bon fonctionnement de l'application. Composer automatise le processus, réduisant ainsi les risques d'erreurs humaines et améliorant l'efficacité du développement.

## Cycle de vie d'une fonctionnalité

- **Définition des besoins :**
- La première étape dans la conception de chaque fonctionnalité est de définir clairement les objectifs et les besoins du système. Pour ce projet, les objectifs principaux sont les suivants :
- **Objectifs principaux du projet :**
- **Formulaire de contact et de retour d'avis :** Permettre aux visiteurs du zoo de contacter le personnel via un formulaire de contact et de laisser un avis sur leur expérience.
- **Gestion des utilisateurs :** Permettre à l'administrateur du zoo de créer des comptes pour les vétérinaires et les employés.
- **Gestion des informations du zoo :** Permettre à l'administrateur de gérer les horaires d'ouverture, les services proposés, les habitats et les animaux du zoo.
- **Gestion des avis et des réponses :** Permettre aux employés du zoo de traiter les avis des visiteurs et d'y répondre par email.
- **Avis médicaux des vétérinaires :** Permettre aux vétérinaires de saisir des avis médicaux sur chaque animal du zoo.
- **Besoins fonctionnels :**
- **Formulaires :**
- Le formulaire de contact recueille des informations telles que le titre, l'email et le message des visiteurs.
- Le formulaire d'avis permet aux visiteurs de donner un retour sur leur visite.
- Ces formulaires doivent être accessibles via des interfaces web simples et fonctionnelles, en utilisant **HTML**, **CSS**, et **Bootstrap** pour garantir une bonne ergonomie et réactivité.
- **Gestion des comptes utilisateurs :**
- Un système de création des comptes pour les **vétérinaires** et les **employés**, avec des rôles et permissions spécifiques pour chaque type d'utilisateur.
- **Gestion des données du zoo :**
- Les informations sur les horaires du zoo, les services, les habitats et les animaux doivent être facilement modifiables via l'interface admin.
- **Gestion des avis des visiteurs et des réponses par email :**
- Les employés du zoo peuvent accéder aux avis des visiteurs et y répondre par email, via un système d'envoi d'emails intégré, tel que **PHPMailer**.
- **Gestion des avis médicaux des vétérinaires :**
- Les vétérinaires peuvent laisser des commentaires sur la santé de chaque animal via un formulaire spécifique.
- **Besoins non fonctionnels :**

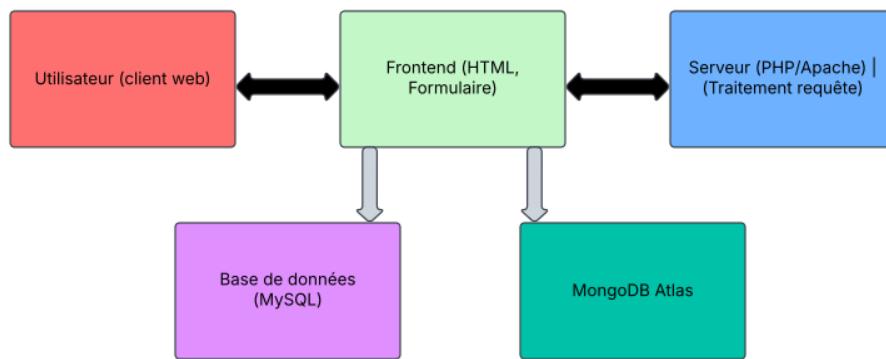
- **Accessibilité et responsive design :** Le site doit être accessible sur tous les types d'appareils (ordinateurs, tablettes, téléphones) grâce à **Bootstrap**.
- **Sécurité des données :** Les informations des visiteurs et des employés doivent être sécurisées, notamment via des mécanismes d'authentification et de gestion des sessions.
- **Performance :** Les pages du site doivent se charger rapidement, même avec un grand nombre de données (animaux, avis, etc.).
- **Architecture de la solution :**
- Une fois les besoins identifiés, il est nécessaire de définir l'architecture technique qui répondra à ces besoins. L'architecture se divise en plusieurs couches, chacune jouant un rôle spécifique dans le bon fonctionnement du projet.
- **Frontend :**
- **Technologies utilisées :**
- **HTML, CSS, JavaScript, et Bootstrap.**
- Les formulaires de contact et d'avis sont créés en **HTML**, avec un design soigné en **CSS** et **Bootstrap** pour assurer un rendu responsive et convivial.
- **JavaScript** est utilisé pour des petites animations comme des slides d'images.
- **Backend :**
- **Technologies utilisées :**
- **PHP** : Langage principal utilisé pour la logique serveur, notamment pour la gestion des formulaires, l'envoi d'emails et la gestion des utilisateurs.
- **MySQL** : Base de données relationnelle pour stocker les informations structurées, comme les comptes utilisateurs, les horaires du zoo, les services, les animaux et les avis des visiteurs.
- **MongoDB** : Base de données NoSQL utilisée pour stocker des données non structurées ou semi-structurées, comme les avis médicaux des vétérinaires.
- **Gestion des emails :**
- **PHPMailer** : Utilisé pour envoyer des emails de confirmation aux visiteurs après leur soumission de formulaire, et pour répondre aux emails des visiteurs. Le tout est testé en local avec **Mailpit**.
- **Diagramme d'architecture :**

Un diagramme pourrait illustrer les interactions entre le frontend (formulaires HTML), le backend (PHP, MySQL, MongoDB), et les services externes comme l'envoi d'emails (PHPMailer) et la gestion des utilisateurs.

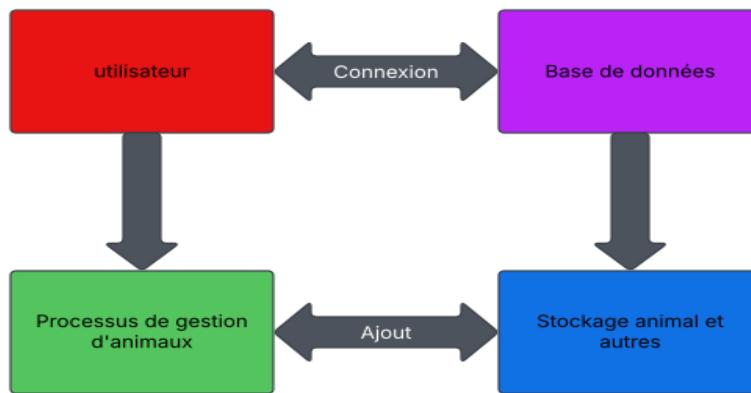
- **Diagrammes et flux**

Des diagrammes peuvent être utiles pour visualiser l'architecture du projet ainsi que les flux de données. Voici quelques exemples :

- **Diagramme de flux des données :**



- **Diagramme d'architecture technique :**



- **Choix technologiques :**

- Voici un aperçu des choix technologiques justifiés pour ce projet :
- **Frontend :**
- **HTML, CSS, Bootstrap, JavaScript** : Ces technologies ont été choisies pour leur efficacité à créer une interface utilisateur propre et responsive.
- **Backend :**
- **PHP** : Langage bien adapté pour le développement web dynamique, notamment pour le traitement des formulaires et l'envoi d'emails.
- **MySQL** : Base de données relationnelle idéale pour gérer les informations structurées telles que les utilisateurs, les horaires du zoo et les services.
- **MongoDB** : Utilisé pour stocker des données non structurées comme les avis médicaux des vétérinaires.
- **Emails :**
- **PHPMailer** : Utilisé pour envoyer des emails via SMTP, offrant une configuration flexible et des fonctionnalités robustes pour l'envoi d'emails transactionnels (confirmation de soumission, réponses aux visiteurs).
- **Mailpit** : Outil utilisé pendant le développement pour tester l'envoi d'emails sans risquer d'envoyer des emails réels.

## Développement frontend

- **Structure HTML, mise en forme CSS et utilisation de Bootstrap pour le design responsive :**
- **Structure HTML :**
- La structure HTML de la page est construite de manière à être claire, fonctionnelle et évolutive. L'architecture utilise l'inclusion de composants externes avec PHP pour simplifier la gestion de la navigation et du footer, ce qui permet une réutilisation efficace du code. Voici les éléments clés de la structure HTML de ton formulaire d'avis :
- **Inclusion des fichiers externes :**
- Le code commence par inclure des fichiers de templates PHP, ce qui permet de centraliser la gestion de l'entête et de la barre de navigation dans des fichiers séparés (`clientHeader.php`, `clientNavbar.php`), ainsi que le pied de page avec `clientFooter.php`. Cela facilite la maintenance du code, car tout changement au niveau de l'entête ou du footer est appliqué automatiquement sur toutes les pages.

```
<?php include 'template/clientHeader.php'; ?>
<?php include 'template/clientNavbar.php'; ?>
```

- **Contenu principal :**
- Le formulaire d'avis est intégré dans une `<div class="container mt-5">`, une structure qui permet de placer correctement le contenu avec un espacement suffisant par rapport aux autres éléments de la page. Le formulaire permet aux utilisateurs de laisser un avis en remplissant un champ pour le pseudo, en sélectionnant une note par étoiles, et en ajoutant un commentaire.

```

<div class="container mt-5">
    <h1 class="text-center">Laissez un avis</h1>

    <!-- Affichage des messages d'erreur et succès -->
    <?php if (isset($successMessage)): ?>
        <div class="alert alert-success mt-4" role="alert">
            <?= htmlspecialchars($successMessage) ?>
        </div>
    <?php elseif (isset($errorMessage)): ?>
        <div class="alert alert-danger mt-4" role="alert">
            <?= htmlspecialchars($errorMessage) ?>
        </div>
    <?php endif; ?>

    <form id="review-form" method="POST" action="/submit-review" class="mt-4">
        <div class="mb-3">
            <label for="pseudo" class="form-label">Pseudo</label>
            <input type="text" class="form-control" id="pseudo" name="pseudo" required placeholder="Votre pseudo">
        </div>

        <div class="mb-3">
            <label class="form-label">Notation</label>
            <div id="rating" class="d-flex align-items-center">
                <input type="radio" id="star5" name="rating" value="5" required>
                <label for="star5" class="star">★</label>

                <input type="radio" id="star4" name="rating" value="4">
                <label for="star4" class="star">★</label>

                <input type="radio" id="star3" name="rating" value="3">
                <label for="star3" class="star">★</label>

                <input type="radio" id="star2" name="rating" value="2">
                <label for="star2" class="star">★</label>

                <input type="radio" id="star1" name="rating" value="1">
                <label for="star1" class="star">★</label>
            </div>
        </div>

        <div class="mb-3">
            <label for="comment" class="form-label">Commentaire</label>
            <textarea class="form-control" id="comment" name="comment" rows="5" required placeholder="Écrivez votre commentaire ici..."></textarea>
        </div>

        <button type="submit" class="btn btn-success">Soumettre l'avis</button>
    </form>
</div>

<?php include 'template/clientFooter.php'; ?>

```

- **Pied de page :**
- Le pied de page est également inclus via un fichier PHP, permettant ainsi de garder un design cohérent à travers toutes les pages du site.
- **Styles de base :**
- Les styles globaux sont définis pour garantir une expérience utilisateur agréable. La police utilisée est '**Roboto**', une police sans-serif moderne et lisible, et le fond est un dégradé de verts pour rappeler l'environnement naturel du zoo.

```
body {
    font-family: 'Roboto', sans-serif;
    margin: 0;
    padding: 0;
    background: linear-gradient(to bottom, #a8e063, #56ab2f) !important;
    color: #3c403d; /* Gris foncé pour le texte */
}
```

- **En-tête et navigation :**
- L'en-tête de la page utilise une barre de navigation qui devient fixe avec la classe **sticky-top**, ajoutant une ombre légère pour la rendre visible même lorsque l'utilisateur fait défiler la page.

```
.sticky-top {
    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1) !important;
    z-index: 1030 !important;
}
```

- **Formulaire d'avis :**
- Le formulaire est centré et chaque champ est stylisé avec la classe Bootstrap **form-control**, ce qui garantit une apparence uniforme et moderne. Le bouton d'envoi a une couleur verte correspondant à la palette du zoo et change légèrement au survol pour un effet interactif.

```
/* Buttons */
button {
    background-color: #2c662d; /* Vert forêt */
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    font-size: 1em;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

button:hover {
    background-color: #1e4620; /* Vert plus foncé */
}
```

- **Étoiles de notation :**
- Les étoiles utilisées pour la notation sont représentées par des éléments **<input type="radio">**, et sont stylisées via CSS pour afficher des étoiles dorées quand elles sont sélectionnées. L'alignement des étoiles est géré avec Flexbox pour les espacer correctement.

```
/* Radio étoiles */
#rating {
    gap: 55px;
    display: flex;
    flex-direction: row-reverse;
    justify-content: flex-end;
}
input {
    display: none;
}
.star {
    font-size: 1.5rem;
    color: gray;
    cursor: pointer;
    transition: color 0.2s;
}
input[type="radio"]:checked ~ label.star {
    color: gold;
}
```

- **Responsivité :**
- Des media queries sont utilisées pour rendre le site responsive. Par exemple, sur les écrans plus petits (tablettes et téléphones), le nombre de colonnes dans les grilles et la taille des polices sont ajustés pour s'adapter à la taille de l'écran.

```
/* Responsiveness */

@media (min-width: 768px) and (max-width: 992px) {
    .carousel-inner {
        position: relative;
        width: 100%;
        overflow: hidden;
        height: 400px;
    }
    .col-md-3 {
        flex: 1 1 calc(33% - 20px) !important;
        max-width: calc(33% - 20px) !important;
    }
    h1, h2 {
        font-size: 2em;
    }
    ul li {
        font-size: 1.2em;
    }
}

@media (min-width: 480px) and (max-width: 768px) {
    .carousel-inner {
        position: relative;
        width: 100%;
        overflow: hidden;
        height: 275px;
    }
}
```

- **Conclusion**
- Cette section présente la manière dont la structure HTML et la mise en forme CSS sont intégrées dans le projet pour créer une interface utilisateur claire, esthétique et responsive. En utilisant PHP pour l'inclusion de templates et en

appliquant un design basé sur des standards modernes avec CSS, ce projet offre une expérience adaptable à différentes tailles d'écran.

## Déploiement

Le déploiement du projet sur Hostinger a impliqué plusieurs étapes, allant de la gestion des fichiers à la configuration des bases de données, en passant par l'intégration du service de messagerie et la gestion des règles de réécriture pour l'URL. Voici un aperçu détaillé du processus suivi :

### **1. Création du compte Hostinger et choix de l'hébergement**

Tout d'abord, un compte a été créé chez Hostinger, et un **hébergement mutualisé** a été sélectionné, ce qui est adapté aux besoins du projet étant donné qu'il ne nécessite pas une infrastructure dédiée ou une très haute performance.

### **2. Gestion des fichiers et upload sur le serveur**

Pour transférer le projet sur le serveur Hostinger, j'ai utilisé leur **gestionnaire de fichiers** intégré, qui permet de facilement uploader les fichiers et dossiers nécessaires au bon fonctionnement du site.

- Les **fichiers HTML, PHP, CSS et JavaScript** ont été transférés dans le répertoire correspondant, tout en respectant la structure du projet.
- Deux fichiers **.htaccess** ont été créés et configurés :
  - **Fichier .htaccess à la racine du projet** : Ce fichier permet de rediriger les requêtes vers le dossier **public** de manière propre. Voici son contenu :
    - `RewriteEngine On`
    - `RewriteRule ^$ public/index.php [L]`
    - `RewriteRule (.*) public/$1 [L]`
  - **Fichier .htaccess dans le dossier public** : Ce fichier est destiné à gérer les règles de réécriture des URL et empêcher l'accès direct à certains fichiers :
    - `Options +FollowSymLinks -Indexes`
    - `RewriteEngine On`
    - 
    - `RewriteCond %{REQUEST_FILENAME} !-d`
    - `RewriteCond %{REQUEST_FILENAME} !-f`

- `RewriteRule ^ index.php [L]`
- Ces fichiers `.htaccess` assurent le bon fonctionnement des URL et une meilleure sécurité du projet.

### 3. Base de données MySQL

Le projet nécessitant une base de données MySQL, voici les étapes suivies pour l'intégration :

- La base de données MySQL a été importée dans l'environnement de base de données de Hostinger via phpMyAdmin.
- Une fois la base de données créée sur le serveur Hostinger, les informations de connexion (nom de la base de données, utilisateur, mot de passe) ont été mises à jour dans le code PHP du projet, dans le fichier de configuration, pour que l'application puisse se connecter correctement.

### 4. Configuration du service de messagerie

Afin de gérer l'envoi et la réception des e-mails via le formulaire de contact, un service de messagerie a été souscrit chez Hostinger. Ensuite, le contrôleur des contacts a été reconfiguré pour utiliser le serveur de messagerie fourni par Hostinger, permettant ainsi à l'application d'envoyer des e-mails en toute sécurité.

Dans le fichier contrôleur ContactController.php :

```
private function sendConfirmationEmail($to, $title, $message) {
    $mail = new PHPMailer(true);

    try {
        // Configuration du serveur SMTP
        $mail->isSMTP();
        $mail->Host = 'smtp.hostinger.com'; // Hôte SMTP de Hostinger (vérifiez sur leur documentation)
        $mail->Port = 587; // Port SMTP recommandé (587 pour une connexion sécurisée)
        $mail->SMTPAuth = true; // Authentification SMTP activée
        $mail->Username = 'zooservices@arcadia.press'; // Votre nouvelle adresse email
        $mail->Password = '*****'; // Votre mot de passe de messagerie

        // Expéditeur et destinataire
        $mail->setFrom('zooservices@arcadia.press', 'Votre Entreprise');
        $mail->addAddress($to);

        // Contenu de l'email
        $mail->isHTML(true);
        $mail->Subject = "Confirmation de votre message : $title";
        $mail->Body = "<p>Merci pour votre message :</p><p><strong>$message</strong></p><p>Nous vous répondrons dans les plus brefs délais.</p>";
        $mail->send();
        return true;
    } catch (Exception $e) {
        // Vous pouvez logger l'erreur ici pour déboguer
        error_log("Erreur lors de l'envoi de l'email : " . $mail->ErrorInfo);
        return false;
    }
}
```

### 5. Déploiement de MongoDB sur MongoDB Atlas

Le projet utilise également MongoDB pour la gestion de certaines données. Plutôt que d'installer MongoDB localement, il a été choisi d'utiliser MongoDB Atlas, un service cloud, pour héberger la base de données.

- Un cluster MongoDB a été créé sur MongoDB Atlas, et les données ont été importées dans ce cluster.
- Le fichier de connexion à MongoDB a été mis à jour pour inclure les informations nécessaires à la connexion au cluster MongoDB Atlas. La chaîne de connexion de MongoDB Atlas a été incluse dans le code PHP du projet pour garantir une connexion sans erreur.

```
<?php  
  
require '../vendor/autoload.php';  
  
class MongoDBConnection {  
    private $client;  
    private $db;  
  
    public function __construct()  
    {  
        // Connexion à MongoDB  
        $this->client = new MongoDB\Client("mongodb+srv://[REDACTED]@cluster0.nittg.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0");  
        L'URL sont corrects  
        $this->db = $this->client->zoo; // Base de données 'zoo'  
    }  
  
    public function getCollection($collectionName)  
    {  
        return $this->db->$collectionName;  
    }  
}
```

- En outre, un port a été autorisé sur MongoDB Atlas pour garantir que les requêtes provenant du serveur Hostinger puissent atteindre la base de données.

## 6. Vérification du site

Une fois les fichiers uploadés, la base de données configurée et MongoDB Atlas connecté, il a été essentiel de tester le site dans son environnement de production pour s'assurer de son bon fonctionnement. Les tests ont inclus :

- Vérification des formulaires : Tester l'envoi d'avis et la réception de messages via le formulaire de contact.
- Test de la connexion à MongoDB Atlas : S'assurer que les données sont correctement insérées et récupérées.
- Vérification de l'envoi d'e-mails : Tester l'envoi de messages via le serveur de mail de Hostinger.

## 7. Conclusion

Le déploiement sur Hostinger a été effectué avec succès en suivant ces étapes. Grâce à l'utilisation de leur gestionnaire de fichiers et à la configuration précise des

services de base de données et de messagerie, le projet est désormais opérationnel et accessible en ligne.

## Sécurité

- Mesures prises pour sécuriser le projet (protection contre les injections SQL, gestion des mots de passe, etc.)

### 1. Protection contre les injections SQL :

L'une des vulnérabilités les plus courantes dans les applications web est l'injection SQL. Pour protéger le projet contre cette attaque, j'ai utilisé des requêtes préparées avec des binds de paramètres (ou "binding") dans toutes les interactions avec la base de données. Cela empêche les attaquants d'injecter du code SQL malveillant dans les requêtes. Voici un exemple de la classe Habitat.php, qui illustre l'utilisation des requêtes préparées pour effectuer des opérations sécurisées sur la base de données :

```
class Habitat {
    private $conn;

    public function __construct() {
        $database = new Database();
        $this->conn = $database->getConnection();
    }

    // Créer un nouvel habitat
    public function create($name, $description, $image) {
        try {
            $query = "INSERT INTO habitats (name, description, images) VALUES (:name, :description, :image)";
            $stmt = $this->conn->prepare($query);

            // Binding des paramètres pour éviter les injections SQL
            $stmt->bindParam(':name', $name);
            $stmt->bindParam(':description', $description);
            $stmt->bindParam(':image', $image);

            // Exécution de la requête
            if ($stmt->execute()) {
                return true;
            } else {
                throw new Exception('Erreur lors de l\'exécution de la requête');
            }
        } catch (Exception $e) {
            echo "Erreur lors de la création de l'habitat : " . $e->getMessage();
            return false;
        }
    }
}
```

Dans cet exemple, les requêtes sont préparées avant d'être exécutées, et les paramètres sont liés à la requête à l'aide de `bindParam`. Cela empêche toute forme d'injection SQL en assurant que les entrées de l'utilisateur sont traitées comme des données et non comme du code SQL exécutable.

## 2. Gestion sécurisée des mots de passe :

La gestion des mots de passe est un aspect fondamental de la sécurité. Pour protéger les mots de passe des utilisateurs, j'ai utilisé la fonction `password_hash()` pour hacher les mots de passe avant de les stocker dans la base de données. Le hachage rend les mots de passe illisibles, ce qui protège les informations sensibles en cas de compromission de la base de données. Voici un extrait du fichier `User.php` où le mot de passe est haché lors de la création d'un nouvel utilisateur :

```
public function create($username, $password, $role) {
    try {
        // Valider le rôle
        $validRoles = ['veterinarian', 'employee', 'admin'];
        if (!in_array($role, $validRoles)) {
            throw new Exception("Rôle invalide.");
        }

        // Préparer la requête d'insertion
        $query = "INSERT INTO users (username, password, role) VALUES (:username, :password, :role)";
        $stmt = $this->conn->prepare($query);

        // Hacher le mot de passe avant de le stocker
        $hashedPassword = password_hash($password, PASSWORD_BCRYPT);

        // Lier les paramètres
        $stmt->bindParam(':username', $username);
        $stmt->bindParam(':password', $hashedPassword);
        $stmt->bindParam(':role', $role);

        // Exécuter la requête
        return $stmt->execute();
    } catch (Exception $e) {
        error_log("Erreur lors de la création de l'utilisateur : " . $e->getMessage());
        return false;
    }
}
```

Ici, la fonction `password_hash()` utilise l'algorithme `bcrypt`, reconnu comme étant sécurisé pour le hachage des mots de passe. Cela permet de protéger les mots de passe en les rendant irréversibles et difficiles à craquer.

## 3. Protection contre les attaques XSS (Cross-Site Scripting) :

Les attaques XSS permettent à un attaquant d'injecter des scripts malveillants dans une page web, ce qui peut compromettre les utilisateurs en leur envoyant des cookies ou en récupérant des informations sensibles. Pour se protéger contre ces

attaques, j'ai utilisé la fonction **htmlspecialchars()** dans les vues pour assainir toutes les données provenant de l'utilisateur (comme les noms d'habitat, les descriptions, et les liens). Cette fonction transforme les caractères spéciaux en entités HTML, ce qui empêche l'exécution de scripts JavaScript. Voici un exemple de la vue des habitats dans **habitats.php** :

```
<div class="container mt-5">
    <h1 class="text-center my-4">Nos Habitats</h1>
    <div class="row">
        <?php foreach ($habitats as $habitat): ?>
        <div class="col-md-4 mb-4">
            <div class="card h-100">
                " style="height: 200px; object-fit: cover;">
                <div class="card-body">
                    <h2 class="card-title"><?= htmlspecialchars($habitat['name']) ?></h2>
                    <p class="card-text"><?= htmlspecialchars($habitat['description']) ?></p>

                    <!-- Liste des animaux -->
                    <?php if (!empty($habitat['animals'])): ?>
                    <ul class="list-group list-group-flush">
                        <?php foreach ($habitat['animals'] as $animal): ?>
                            <li class="list-group-item">
                                <a class="animal" href="/animal?id=<?= htmlspecialchars($animal['id']) ?>"><strong><?= htmlspecialchars($animal['name']) ?></strong> (<?=
                                htmlspecialchars($animal['breed']) ?>)</a>
                            </li>
                        <?php endforeach; ?>
                    </ul>
                    <?php else: ?>
                        <p class="text-muted mt-3">Aucun animal dans cet habitat.</p>
                    <?php endif; ?>
                </div>
            </div>
        </div>
        <?php endforeach; ?>
    </div>
</div>
```

Dans ce code, **htmlspecialchars()** est utilisé pour assainir les variables **\$habitat[ 'name' ]**, **\$habitat[ 'description' ]**, et **\$animal[ 'name' ]**. Cela empêche l'injection de code JavaScript ou de HTML malveillant dans les pages vues par les utilisateurs.

## Conclusion :

Les mesures de sécurité mises en place dans ce projet permettent de se protéger contre les principales vulnérabilités telles que les injections SQL, les attaques XSS et la gestion des mots de passe. En utilisant des requêtes préparées, en hachant les mots de passe, et en assainissant les données utilisateur, le projet est sécurisé contre des attaques courantes qui peuvent compromettre la sécurité des utilisateurs et des données.

## Problèmes rencontrés et solutions

Durant le développement du projet, plusieurs défis ont été rencontrés, principalement liés aux requêtes SQL, à l'intégration de la bibliothèque MongoDB et à des conflits de style côté frontend. Voici un résumé des principaux problèmes rencontrés et des solutions apportées.

## 1. Erreurs d'écriture des requêtes SQL et doublons d'éléments affichés :

### Problème :

Des erreurs dans l'écriture des requêtes SQL ont provoqué des affichages incorrects dans le frontend, notamment l'apparition de **doublons d'éléments** dans les résultats des requêtes. Cela était lié à la récupération des habitats et des animaux, où la logique de jointure ou les clauses SQL étaient mal formulées.

### Solution :

Pour résoudre ce problème, j'ai analysé et corrigé les requêtes en vérifiant notamment les **jointures** et en m'assurant que chaque requête SQL renvoyait les résultats corrects. En parallèle, j'ai utilisé des outils comme **phpMyAdmin** pour tester mes requêtes avant de les implémenter dans le code PHP. Voici quelques solutions mises en place :

- Utilisation de **requêtes préparées** pour éviter les erreurs de syntaxe.
- Vérification des **clés primaires et étrangères** pour garantir que les données étaient correctement liées.
- Ajout de filtres dans les requêtes pour s'assurer que les doublons étaient évités, notamment avec l'utilisation de **DISTINCT** ou de conditions supplémentaires dans les **WHERE**.

## 2. Conflits entre les éléments frontend et Bootstrap :

### Problème :

Certaines sections du frontend (par exemple, les éléments de grille et de carte) entraient en conflit avec les **classes Bootstrap**, entraînant des affichages incohérents ou des éléments qui ne s'alignaient pas correctement.

### Solution :

Pour résoudre ce problème, j'ai procédé à une révision des éléments HTML et CSS en suivant les bonnes pratiques de **structure Bootstrap**. J'ai aussi utilisé des outils comme **Chrome DevTools** pour inspecter le code et détecter les conflits de styles. Les principales actions entreprises ont été :

- Vérification de la hiérarchie des classes CSS et ajout de classes supplémentaires pour affiner le style.
- Utilisation de la fonctionnalité "**grid**" de Bootstrap pour mieux gérer l'affichage sur différents écrans.

- Ajout de styles personnalisés dans un fichier CSS dédié pour ajuster certains éléments sans perturber les styles globaux.

### 3. Problèmes avec l'extension MongoDB (PECL et php.ini) :

#### Problème :

Lors de l'intégration de **MongoDB** dans le projet, un problème est survenu avec l'extension MongoDB dans le fichier **php.ini**. Le serveur local n'avait pas l'extension MongoDB nécessaire installée, ce qui a empêché la connexion à la base de données MongoDB.

#### Solution :

La solution à ce problème a consisté à installer l'extension **MongoDB pour PHP** via **PECL**, puis à modifier le fichier **php.ini** pour activer cette extension. Voici les étapes suivies :

1. Installation de l'extension MongoDB avec la commande `pecl install mongodb`.
2. Ajout de la ligne `extension=mongodb.so` dans le fichier **php.ini** pour activer l'extension.
3. Redémarrage du serveur PHP pour appliquer les changements.

J'ai également consulté les **docs officielles de MongoDB** pour m'assurer que j'avais bien configuré l'extension et le fichier **php.ini**.

### 4. Recherche de solutions et collaboration avec la communauté

#### Solution générale :

Pour résoudre ces problèmes techniques, j'ai eu recours à plusieurs sources d'informations et de support communautaire :

- **Stack Overflow** : J'ai posé des questions sur ce forum et trouvé des solutions à de nombreux problèmes liés aux requêtes SQL et à MongoDB.
- **Documentation officielle** des bibliothèques utilisées : Les documents de MongoDB, PHP, et Bootstrap ont été consultés pour comprendre les meilleures pratiques et résoudre les problèmes.
- **Vidéos YouTube** : Certaines vidéos tutorielles ont été très utiles pour mieux comprendre l'intégration de MongoDB et la gestion des extensions PHP.
- **Communautés de développeurs** : J'ai sollicité l'aide de forums comme Grafikart ou des serveurs Discord spécialisés pour obtenir des conseils et résoudre rapidement des problèmes bloquants.

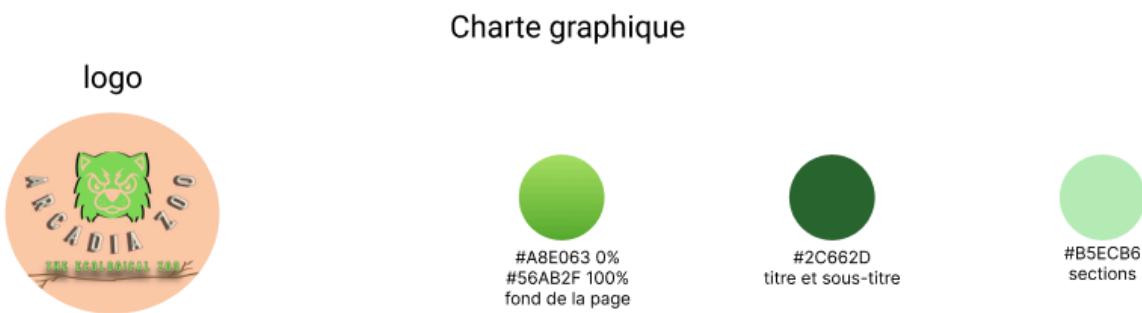
---

#### Conclusion :

Les problèmes rencontrés pendant le développement étaient principalement liés à des erreurs dans les requêtes SQL, des conflits CSS dans le frontend, et des configurations d'extensions pour MongoDB. Grâce à une recherche active sur des forums techniques, des vidéos et des docs officielles, ces défis ont été résolus. La collaboration avec la communauté de développeurs a également été précieuse pour surmonter ces obstacles. Ces expériences m'ont permis de renforcer ma capacité à résoudre des problèmes techniques de manière autonome tout en utilisant les ressources disponibles en ligne.

## Annexe

Charte graphique figma :



Police d'écriture :

Roboto, sans-serif

MAJUSCULE  
minuscule  
0123456789



Wireframes figma :



Mock up figma :

# Acradia Eliess Kreir

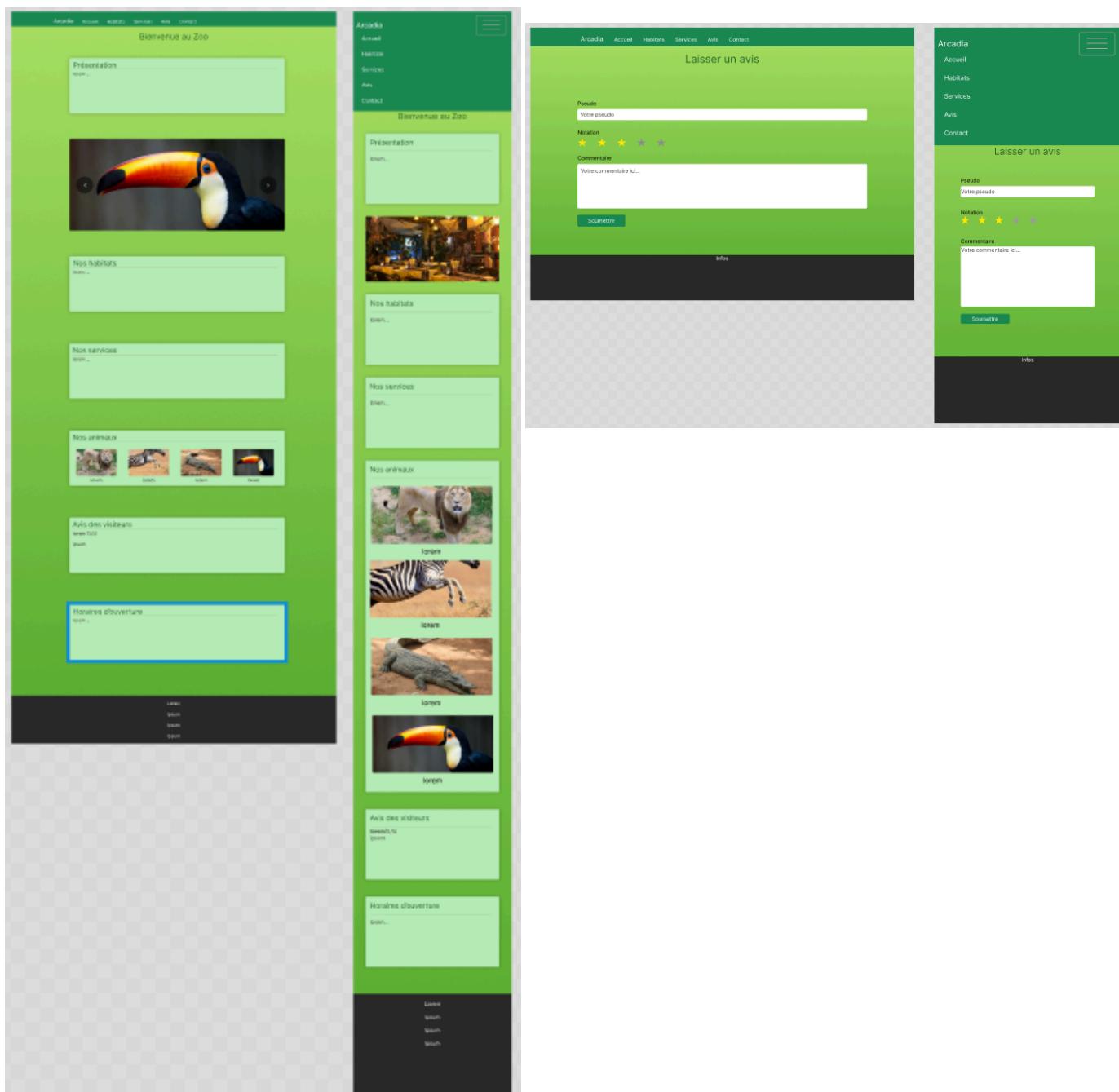


Diagramme UML :

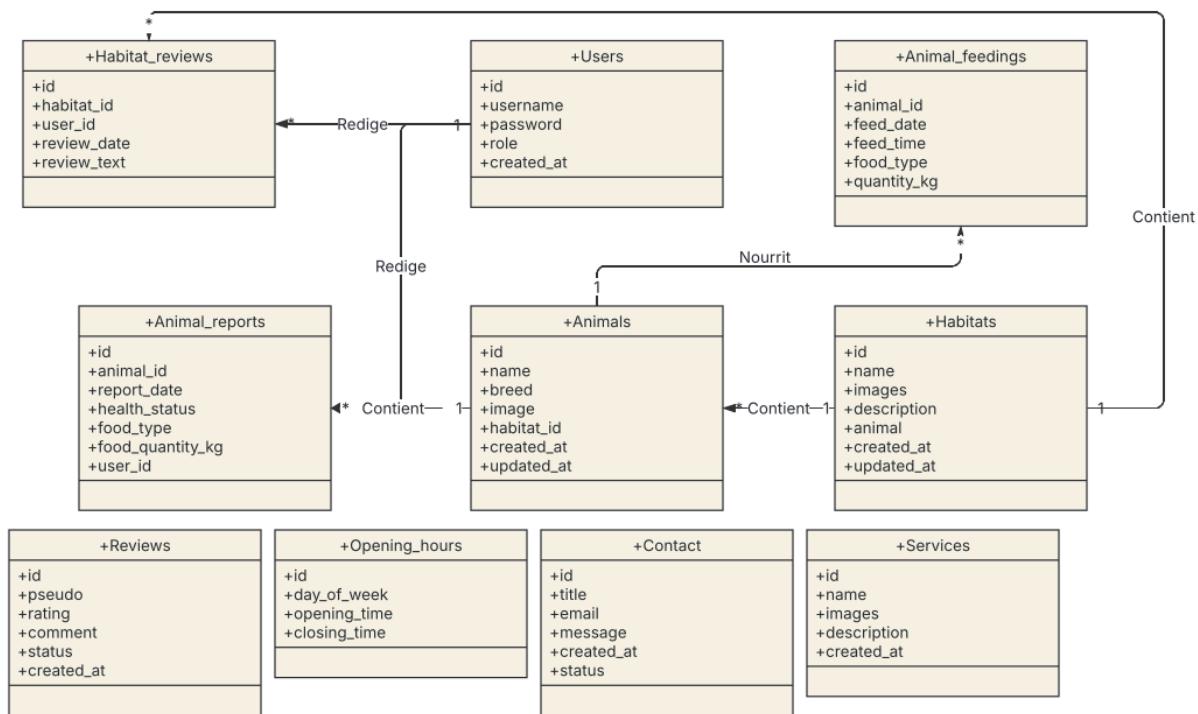


Diagramme de séquence de connexion :

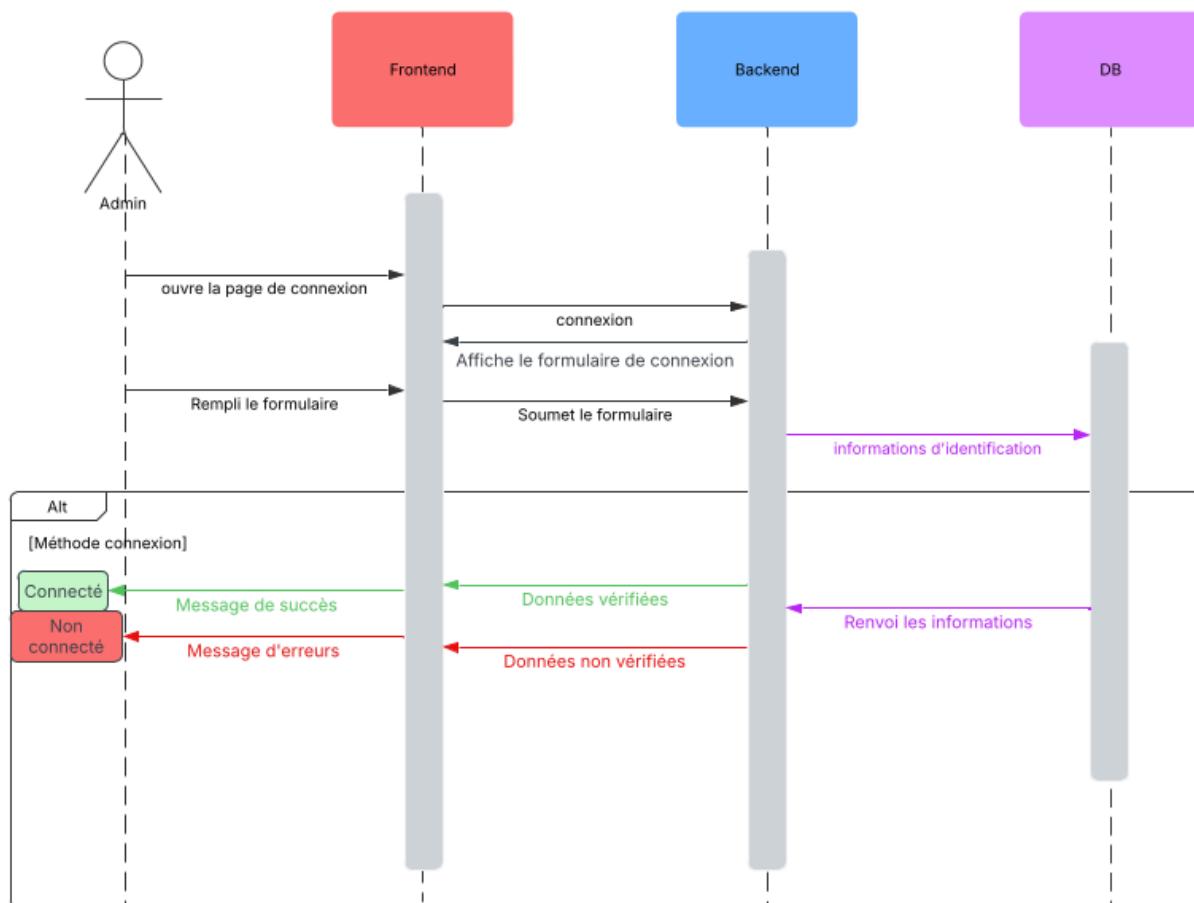
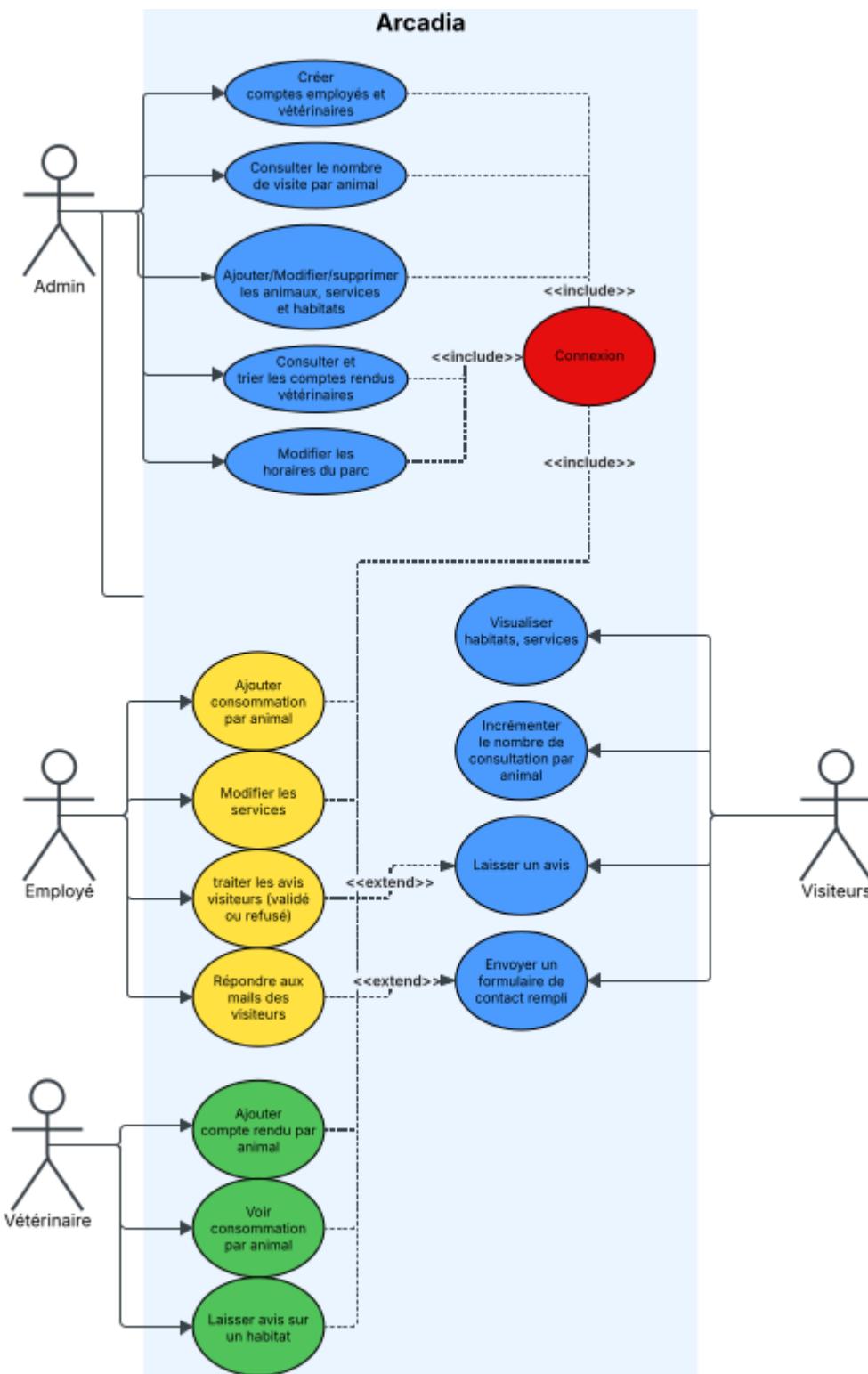


Diagramme de cas d'utilisation :



Gestion de projet et tâches :

The screenshot displays a list of five completed tasks (Terminé) in a dark-themed application:

- Maintenance du site**  
Eliess  
Maintenance et amélioration du site  
Moyenne  
1 comment
- Déploiement**  
Eliess  
Déployer le site web  
Haute  
1 comment
- Développer le site web**  
Eliess  
Développer le site web  
Haute  
1 comment
- Vérification de l'environnement**  
Eliess  
préparer l'environnement de développement  
Haute  
1 comment
- Maquettage du site**  
Eliess  
Terminer la structure et le design du site  
Haute

A button for "Nouvelle tâche" (+ Nouvelle tâche) is visible at the bottom left.

The screenshot shows a project management interface with a dark theme. On the left, there's a sidebar with a tree view of tasks. A specific task, "Développer le site web", is selected and shown in detail on the right.

**Task Details:**

- Titre:** Développer le site web
- Personne assignée:** Eliess
- État:** Terminé
- Résumé:** Développer le site web et tester les fonctionnalités
- Échéance:** 10 octobre 2024 → 24 octobre 2024
- Projet:** Développer le site web
- Priorité:** Haute
- Étiquettes:** Mobile, Site Web

**Propriétés:** 1 autre propriété

**Relations:** Ajouter Sous-tâches

**Commentaires:**

- Eliess 14/11/2024 (modifié)  
Fait, structure, sécurité et fonctionnalité du site sont opérationnelles.
- Eliess Ajouter un commentaire...

**Description:**

- Créer la base de donnée et les tables
- Mettre ~~git~~ en place