Documentation technique

1. Réflexions initiales technologiques sur le sujet

Trello : La méthode Kanban étant requise et dans l'objectif de minimiser les couts, le logiciel Trello comme logiciel de gestion de projet s'est tout de suite imposé.

Figma : La nécessité de faire ressortir l'aspect écologique du zoo, a amené à choisir les couleurs évoquant la nature des différents habitats. Ces couleurs sont recensées dans la chartre graphique. Nous avons opté pour l'application Figma afin de réaliser les maquettes. Elle permet de prédéfinir ces couleurs ainsi que les fontes et permet la définition de composants réutilisables, qui rendent la conception plus rapide et efficiente.

Lopping : Le choix du progiciel Lopping pour la définition du MCD. Il nous a paru très fonctionnel, par rapport à sont cout (il est gratuit).

Draw.io : Pour les graphes, Draw.io est un bon compromis entre aspect visuel et facilité de réalisation.

PostgreSql : Pour les données, nous avons opté pour une base de type relationnel. Elle permet une bonne cohérence des différentes entités et permet de les requêter très facilement, en utilisant les avantages du langage SQL qui permet de joindre très facilement les différentes tables.

MongoDb : Cependant, la nécessité de disposer d'un Dashboard comprenant des graphes sur les consultations d'animaux par les visiteurs nous a amené à choisir le type de base de données non-relationnel de type NoSql pour les gérer. En effet ces actions représentent une forte volumétrie et des temps d'agrégation très conséquents.

Framework Angular 17 : Pour l'application elle-même, nécessitant légèreté et rapidité, le choix d'une architecture client-serveur avec un front-end de type STA s'imposait. Le mode STA permet de charger sur le navigateur l'ensemble des composants graphiques et des scripts de traitement, nécessaires au front-end. Et cela, dès le lancement de l'application. C'est

couteux au lancement, mais une fois chargée, l'application est de très loin beaucoup plus rapide.

Framework Spring Boot – Java : pour l'API RESTFul du back-end. La robustesse de Java assure des applications plus fiables et résistantes.

JWT : Les espaces spécifiques pour certaines catégories de personnes (administrateur, vétérinaires et employés) devant être gérés avec une authentification, la technique du JWT (Json Web Token) est la plus adaptée.

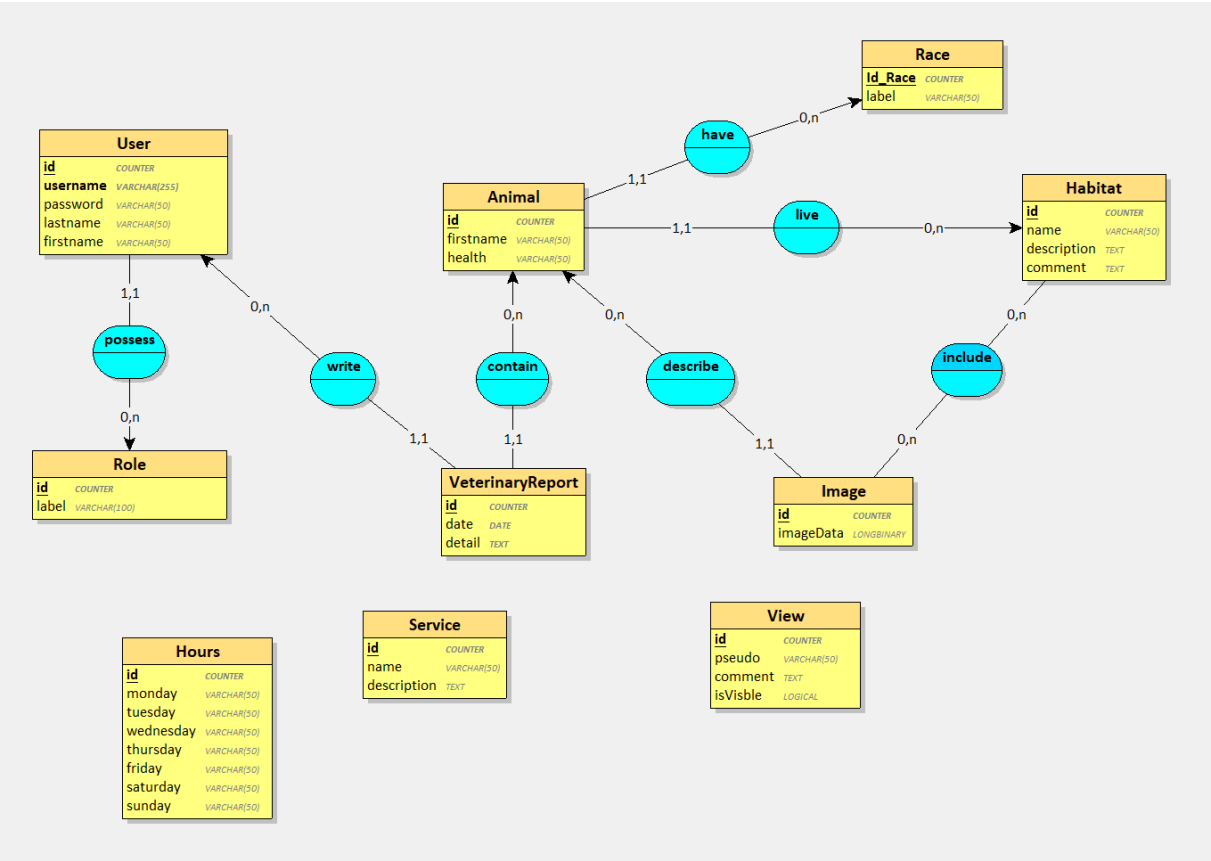
Github : Pour le versionning et le stockage des composants développés, le système Git a été choisi. C'est le plus connu et le plus performant. Il permet le versionning sous forme de commit qui sont des captures de l'application a des instants donnés. Son système de branches permet également à plusieurs fonctionnalités d'être développées par une ou plusieurs personnes. La plateforme **Github**, par sa grande communauté de développeurs et des services de déploiement automatiques, nous a paru la plus adaptée.

OVH : Pour l'hébergement, un VPS (Serveur privé virtuel) de la société française OVH est du meilleur rapport performance/prix du marché.

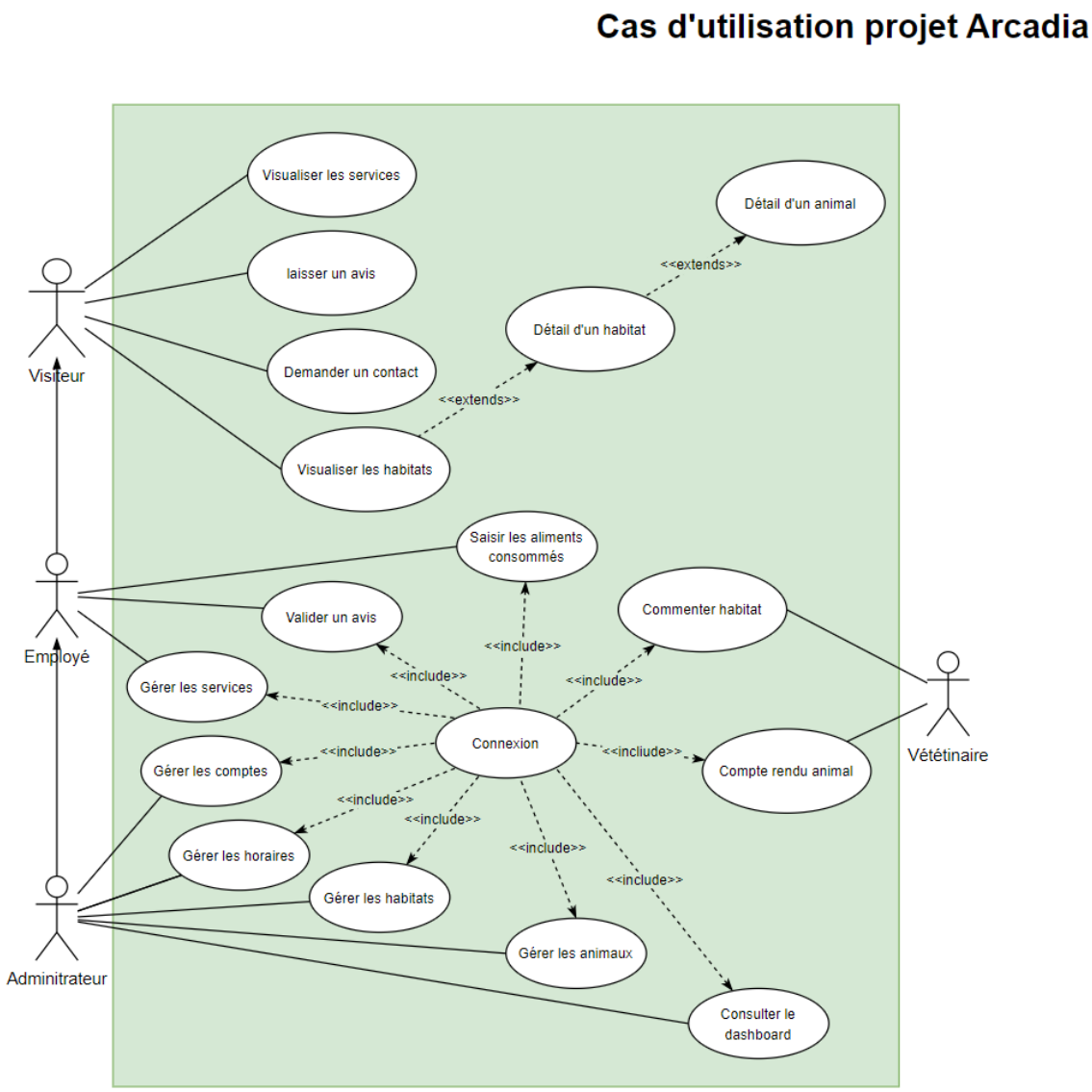
2. Configuration de l'environnement de travail

- **Trello** : création d'une application Arcadia
- **Github** : Création de deux repositories. Un pour le front-end et un pour le back-end.
- **OVH** : souscription d'un VPS
- **Visual Studio Code** pour l'édition du code, avec les extensions :
    - **Angular Language Service** : pour la complétion de code et les liens vers la définition des éléments de code
    - **Extension Pack for Java :** pour la fonction de lint, la complétion de code …
    - **Maven for Java** : pour builder application
    - **Git Graph** : propose un graphe des branches plus lisible.
    - **Github Actions** : pour la détection d'erreur de code.
    - **Draw.io Integration** : pour les graphes de conception.
- **Docker** et **docker desktop** : pour les environnements de demo et de prod
- **DBeaver** : comme outil de de la base relationnelle PostgreSql.
- **MongoDBCompass** : pour la gestion de la base Mongodb.
- **Looping** : comme interface graphique de création du MCD.
- **Postman** : pour les tests du back-end.

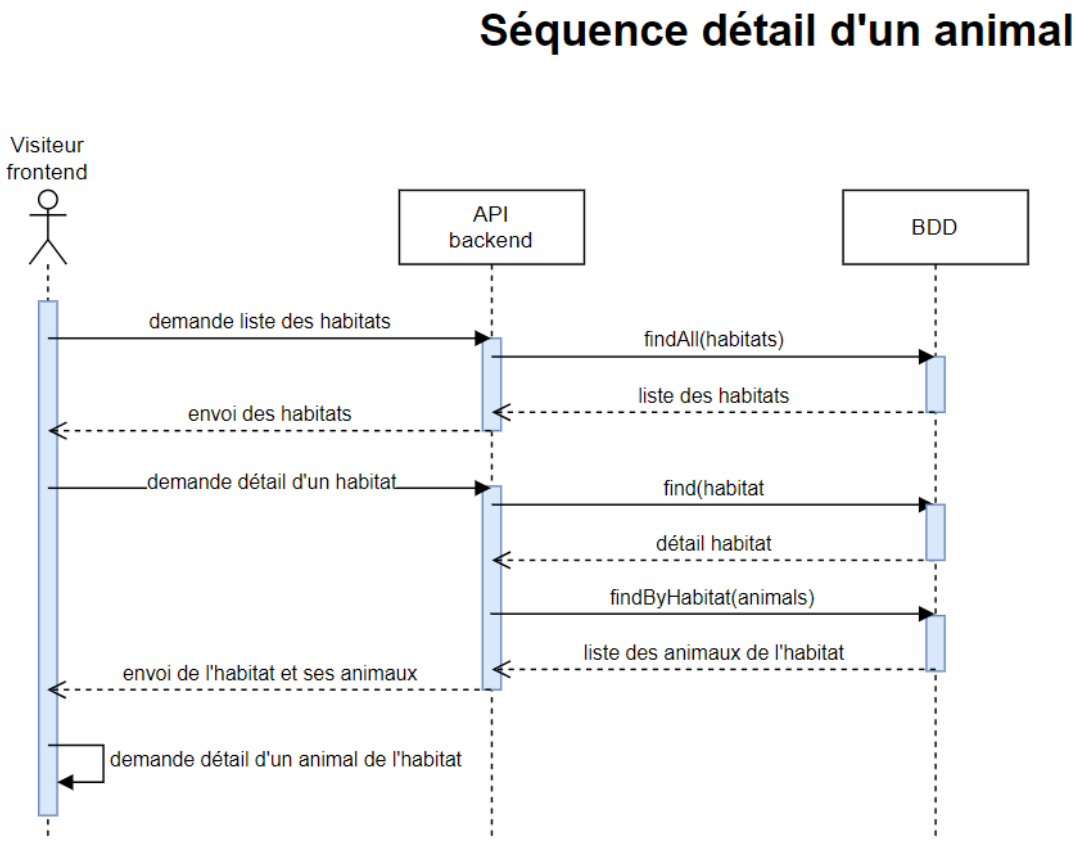
3. Modèle conceptuel de données



4. Diagrammes de cas d'utilisation



5. Diagramme de séquence



6. Documentation du déploiement

a. Introduction

Le déploiement en production, comme le stockage du projet sur Github, isole la partie back-end de la partie front-end. Ces 2 parties sont gérées comme 2 applications indépendantes, qui communiquent entre elles grâce à leur seul paramétrage. Pour les deux parties, les technologies utilisées sont :

- Repository Github pour le stockage des sources.
    - Un repository pour chaque partie.
    - Une branche **dev** pour les développements.
    - Une branche **main** pour le déploiement en production.
    - Le merge de dev sur main déclenche le Github Actions, par la simple présence d'un fichier .yml dans le répertoire **.github**

- Docker et docker compose pour le montage des serveurs
    - Un fichier **docker-compose.yml** pour chaque partie.

- VPS OVH pour l'hébergement
    - Un même serveur héberge les deux parties dans des containers dockers différents
    - Pour l'url, un domaine a été créé chez Namecheap qui fournit les certificats pour la partie TLS. Le backend écoute sur le port 8443 et le front-end sur 443

b. Dockerfile et docker compose

Un fichier Dockerfile par partie permet la création des images. Ils utilisent chacun deux images docker, une pour le build et une pour le container final. Chaque image est créée par docker compose.

i. Back-end

Le fichier Dockerfile utilise l'image **maven:3.9.7-eclipse-temurin-21-jammy** pour builder l'application. Le fichier war généré, ainsi que le fichier des certificats sont copiés ensuite vers une image créée à partir de l'image **tomcat:jre21-temurin-jammy.** Cette image étant l'image finale du back-end.

Pour créer le container du backend et ceux des bases associées, un fichier docker-compose.yml crée trois services :

- Un pour le back à partir d'une image générée par Github action lors du déploiement, et stockée dans le hub Github : **ghcr.io/etupdt/ecf-arcadia-back:latest**. Cette image de GitHub est elle-même générée à partir du Dockerfile. Ce service comprend un volume persistant pour les images
- Un pour la base de données Postgresql à partir de l'image : **postgres:14-alpine**. Ce service comprend un volume persistant pour les données.
- Un pour la base de données Mongodbà partir de l'image **mongo:8.0.0-rc8-jammy**. Ce service comprend un volume persistant pour les données.

ii. Front-end

Le fichier Dockerfile utilise l'image **node:20-alpine3.20** pour builder l'application. Les éléments buildés, ainsi que les fichiers des certificats sont copiés ensuite vers une image créée à partir de l'image **nginx:alpine3.19.** Cette image étant l'image finale du front-end.

Pour créer le container du front-end, un fichier docker-compose.yml crée un service :

- A partir d’une image générée par Github Actions lors du déploiement, et stockée dans le hub Github : **ghcr.io/etupdt/ecf-arcadia-front:latest**. Cette image de GitHub est elle-même générée à partir du Dockerfile.

c. Déploiement par Github Actions :

On utilise Github Actions lors du déploiement en production des deux parties de l'application. Pour chacune d'elles, un fichier deploy.yml défini les tâches à exécuter. Ces tâches s'appuient sur deux dictionnaires : un de « secrets » qui sont des variables dont les valeurs sont saisies dans Github et cryptées pour ne plus être accessibles que du processus de déploiement. Et un de « variables » qui est la même chose mais non crypté pour des valeurs non sensibles. Le déploiement Github Actions initié par le merge sur la branche main exécute les tâches du fichier yml.

i. Back-end

1. Github Actions met à disposition un serveur temporaire en vue de builder l'image du back-end à partir du dockerfile.
2. Il récupère la branche main du repository.
3. Il génère le fichier env.properties à partir des magasins de « secrets » et de « variables » Github.
4. Il build l'image Docker du back-end et la push sur le hub Github.
5. Il copie alors le fichier env.properties vers le serveur d'hébergement OVH.
6. Il copie également les fichiers d'initialisation des bases Postgresql et mongodb vers le serveur d'hébergement OVH.
7. Il copie le fichier docker compose vers le serveur d'hébergement OVH.
8. Il lance le pull de l'image du hub docker sur le serveur d'hébergement OVH.
9. Il crée les container Docker à partir de cette image sur le serveur d'hébergement OVH.

10. Les containers back-end, PostgreSql et Mongodb sont alors opérationnels sur le serveur OVH

ii. Front-end

1. Github Actions met à disposition un serveur temporaire en vue de builder l'image du front-end à partir du dockerfile.
2. Il récupère la branche main du repository.
3. Il build l'image Docker du front-end et la push sur le hub Github.
4. Il copie le fichier docker compose vers le serveur d'hébergement OVH.
5. Il lance le pull de l'image du hub docker sur le serveur d'hébergement OVH.
6. Il crée les container Docker à partir de cette image sur le serveur d'hébergement OVH.
7. Le container front-end est alors opérationnel sur le serveur OVH