

**Dossier projet**

# **Titre Professionnel DWWM (RNCP 37674)**

**Graduate Developer Angular**

## **Zoo Arcadia**

Thomas Philipps  
né le 25/05/1978

**Session de novembre/décembre 2024**



# Sommaire

	Page
• Liens Internet – Accès au site	4
• Introduction	
◦ Objectifs de l'ECF	5
◦ Résumé du projet	5
• Cahier des charges	6
• Préparation du projet	8
• Documentation technique	
◦ Stack technique	9
◦ Configuration de l'environnement de travail	10
◦ Diagrammes UML	
▪ Diagramme de classe	12
▪ Diagramme de cas d'utilisation	12
▪ Diagrammes de séquence	12
◦ Déploiement	13
• Gestion du projet	
◦ Diagrammes	16
◦ Recherche graphique	
▪ Charte graphique	16
▪ Mockups et Wireframes	16
◦ Préparation du développement	17
◦ Back-end	18
◦ Front-end	19
• Sécurité	24
• Recherche anglophone	29
• Conclusion	30
• Annexes	31



## Liens Internet du projet

Application déployée :

<https://zoo-arcadia-tp.netlify.app/home>

Dépôt Github :

<https://github.com/thomasphilipps/arcadia>

Gestionnaire de projet :

<https://thomas-philipps.notion.site>

## Identifiants

**Transmis uniquement aux correcteurs du projet**

**Sur demande par message sur le dépôt GitHub**

# Introduction

## Objectifs

Dans le cadre de ma formation en tant que Développeur Web et Web Mobile chez STUDI, un projet couvrant tous les objectifs nécessaires à l'obtention du Titre Professionnel (TP) nous a été proposé.

C'est donc dans ce cadre que j'ai réalisé ce projet d'application web.

Les objectifs visés par le TP sont définis comme suit :

### 1. Développer la partie front-end d'une application web ou web mobile sécurisée

- Installer et configurer son environnement de travail en fonction du projet web ou web mobile
- Maquetter des interfaces utilisateur web ou web mobile
- Réaliser des interfaces utilisateur statiques web ou web mobile
- Développer la partie dynamique des interfaces utilisateur web ou web mobile

### 2. Développer la partie back-end d'une application web ou web mobile sécurisée

- Mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL et NoSQL
- Développer des composants métier coté serveur
- Documenter le déploiement d'une application dynamique web ou web mobile

## Résumé du projet

Le projet d'application web proposé est le suivant :

« Arcadia est un zoo situé en France près de la forêt de Brocéliande, en Bretagne depuis 1960.

Ils possèdent tout un panel d'animaux, réparti par habitat (savane, jungle, marais) et font extrêmement attention à leur santé. Chaque jour, plusieurs vétérinaires viennent afin d'effectuer les contrôles sur chaque animal avant l'ouverture du zoo afin de s'assurer que tout se passe bien, de même, toute la nourriture donnée est calculée afin d'avoir le bon grammage (le bon grammage est précisé dans le rapport du vétérinaire).

Le zoo, se porte très bien financièrement, les animaux sont heureux. Cela fait la fierté de son directeur, José, qui a de grandes ambitions.

A ce jour, l'informatique et lui ça fait deux, mais, il a envie d'une application web qui permettrait aux visiteurs de visualiser les animaux, leurs états et visualiser les services ainsi que les horaires du zoo.

Ne sachant pas comment faire, José a demandé à Josette, son assistante, de trouver une entreprise lui permettant d'obtenir cette application et augmenter ainsi la notoriété et l'image de marque du zoo.

Après l'obtention de votre diplôme, vous avez été embauché au sein de DevSoft, entreprise qui a été choisi par Josette afin de réaliser l'application. »

# Cahier des charges

L'application devra comporter les éléments suivants :

- **Page d'accueil**

Elle devra intégrer une présentation rapide du zoo en incluant quelques images, mentionner les différents habitats, services et animaux pensionnaires du zoo, et présenter les avis des visiteurs du zoo

- **Menu de l'application**

Il devra au minimum comporter

- Un retour vers la page d'accueil
- Un accès vers le détail des services
- Un accès vers le détail des habitats
- Un accès vers une page de contact
- La connexion à l'espace d'administration pour les vétérinaires, employés et administrateurs

- **Détail des services**

Une interface simple pour récapituler tous les services que propose le parc. Les services doivent comporter au minimum un nom et une description.

Les services actuellement disponibles sont : Restauration, visite guidée gratuite et visite du zoo en petit train.

- **Détail des habitats**

Cette page doit mentionner tous les habitats du zoo ainsi que les animaux associés.

Un habitat est caractérisé par un nom, une ou plusieurs images, une description.

Un animal est caractérisé par un prénom, une espèce, une ou plusieurs images, l'habitat d'affectation.

Au clic sur un animal, on doit pouvoir avoir accès au détails le concernant, ainsi que son état renseigné par les vétérinaires du zoo (voir partie administration de ce cahier des charges)

- **Avis des visiteurs**

Un visiteur doit pouvoir laisser un avis qui sera modéré par un employé du zoo. Cet avis comportera le nom ou pseudo ainsi qu'un champ texte pour s'exprimer. Les employés pourront également ajouter eux même des avis.

- **Espace d'administration**

L'espace d'administration est accessible à l'administrateur du site, aux employés et aux vétérinaires

- *Administrateur*

L'administrateur est unique.

Ce compte devra être créé manuellement lors du déploiement du site.

L'administrateur doit pouvoir effectuer les tâches suivantes :

- Créer des employés et des vétérinaires :  
Le login sera l'adresse e-mail de la personne, et l'administrateur choisira un mot de passe. Lors de la création de son accès, l'utilisateur recevra un courriel l'invitant à se rapprocher de l'administrateur pour obtenir son mot de passe.
- Créer, modifier et supprimer les services, horaires, habitats et animaux du zoo
- Voir les comptes rendus vétérinaires avec possibilité de filtrer les résultats

L'administrateur doit également avoir accès à un dashboard lui permettant de consulter les statistiques de consultations par animal

- *Employé*

Les employés doivent, depuis leur espace, pouvoir effectuer les tâches suivantes :

- Valider ou invalider un avis. Les avis validés seront alors visibles sur la page d'accueil.
- Modifier les services du zoo
- Enregistrer l'alimentation quotidienne des animaux selon les préconisations des vétérinaires

- *Vétérinaire*

Les vétérinaires passent quotidiennement dans le zoo et remplissent un compte rendu de visite. Ce dernier comprend, pour chaque animal :

- L'état général de l'animal
- La nourriture proposée (type et grammage)
- Date de passage
- Un détail de l'état de l'animal (information facultative)

Les vétérinaires peuvent également commenter l'état des habitats afin de voir s'il peut être amélioré.

Enfin, le vétérinaire peut consulter la nourriture donnée à chaque animal.

- **Contact**

Un visiteur peut contacter le zoo s'il le souhaite via un formulaire comprenant titre, description et courriel de contact. La demande est envoyée par mail au zoo, la réponse se fait via ce moyen.

- **Statistiques**

Chaque consultation d'animal entraîne son enregistrement dans une base de données non relationnelle et devra pouvoir être exploitée dans le dashboard administrateur.

- **Déploiement**

L'application devra être déployée par les soins du développeur.



## Préparation du projet

J'ai effectué ma formation en parallèle de ma vie personnelle et professionnelle.

Étant employé à plein temps, travaillant en horaires décalés et étant absent de mon domicile plusieurs soirs par semaine, il me fallait donc une préparation rigoureuse afin de mener ce projet à bien.

Je devais pouvoir suivre mon projet aussi bien à domicile qu'en déplacement, j'ai donc choisi d'utiliser des outils en ligne et un environnement de développement commun pour le PC fixe que j'utiliserai à domicile.

Dans ce cadre, l'utilisation de Docker s'imposait afin de ne pas avoir de problème de compatibilité entre les versions des différents logiciels que j'utilisais pour mon back-end.

Le plus gros de mon travail préparatoire a été d'intégrer les notions nécessaires au développement d'un serveur NodeJS. En effet, bien qu'ayant des connaissances en JS, la mise en place du serveur et la communication entre les différents éléments étaient des notions qui m'étaient inconnues. Heureusement, de nombreuses sources sont disponibles sur le web hormis la documentation officielle de NodeJS, notamment sur StackOverflow, Medium et YouTube.

J'ai dû également consulté très régulièrement la documentation d'Angular. Bien qu'enseigné par STUDI, ce framework évolue rapidement, les versions majeures étant déployées deux fois par an environ. Par ce fait, je devrais adapter mes connaissances afin que mon application respecte l'évolution d'Angular et les préconisations de sa dev team. C'est ainsi que j'ai décidé notamment d'utiliser les « standalone components » et la nouvelle syntaxe de contrôle de flux (@if, @for plutôt que \*ngIf et \*ngFor par exemple)

Concernant mon organisation personnelle, vu mes contraintes d'emploi du temps, je travaillerai à des horaires irréguliers, sans savoir si combien de temps par jour je pourrai consacrer au projet.

# Documentation Technique

## Stack technique

Bien que le projet ne demande pas de fonctionnalités poussées, j'ai choisi de travailler avec une MEAN stack (MongoDB, Express, Angular, NodeJS) à laquelle j'ai ajouté MySQL pour gérer les bases de données relationnelles.

Les raisons sont multiples:

- Personnelles d'abord, car en m'engageant dans ma formation je souhaitais principalement pouvoir apprendre à travailler avec le langage TypeScript, qui est le langage utilisé par le framework Angular.
- Sécuritaires ensuite. Angular intègre nativement des sécurités pour éviter de nombreux types d'attaques. Ces protections sont décrites au chapitre 7 de ce rapport.
- Ayant déjà effectué quelques projets en JavaScript, l'utilisation de NodeJS pour la partie back-end me permet d'approfondir également ma méthodologie dans ce langage.

Il en découle le stack technique suivant :

### **Front-end**

- **Angular 17** : Framework front-end TypeScript
- **Bulma** : Framework CSS/SCSS pour le design responsive
- **Material Angular** : Composants UI spécialement adaptés à Angular

### **Back-end :**

- **NodeJS** : Serveur JavaScript
- **Express** : Framework pour NodeJS conçu pour simplifier le développement de serveurs Web et d'API
- **Sequelize** : Object Relational Mapping (ORM) facilitant la communication entre NodeJS et les bases de données SQL
- **Mongoose** : ORM facilitant la communication entre NodeJS et les bases de données NoSQL

### **Bases de données :**

- **MySQL** : Base de données relationnelles SQL
- **MongoDB** : Base de données NoSQL

## Stockage externe :

Afin de stocker les images du site, ne sachant pas initialement quelle quantité de données seront utilisées et ne voulant pas inutilement surcharger le serveur, j'ai choisi de les stocker dans un conteneur **Amazon Web Services S3**.

## Service d'e-mail :

Pour la démonstration de déploiement, j'ai créé une adresse **Google Mail**.

## Configuration de l'environnement de travail

Concernant mon environnement de travail, je devais pouvoir travailler aussi bien de mon PC fixe que de mon PC portable, en ayant des environnements identiques afin d'éviter des bugs liés à des versions logicielles différentes.

### 1. Planification : **Notion**

Bien que travaillant de manière régulière et en solo sur ce projet, un logiciel de planification de tâches était nécessaire. J'ai choisi d'utiliser Notion, qui me permet avec une interface très ergonomique de suivre l'avancement de mon travail.

J'ai créé un tableau Kanban en plusieurs parties :

- DevOps pour suivre l'avancement notamment de tous les aspects préparatoires d'un projet : diagrammes UML, charte graphique...
- Back-end
- Front-end

### 2. Versionning : **GIT et GitHub**

Afin de pouvoir travailler en suivant les bonnes pratiques de développement collaboratif, j'ai choisi d'utiliser comme logiciel de versioning GIT.

La branche « master » contient la dernière version fonctionnelle du code, pouvant être mise en production, tandis que la branche « development » fusionnera toutes les fonctionnalités en cours de développement. Une fois testée, la branche « development » est merge sur la branche « master »

Afin de pouvoir travailler sur mes deux machines, j'ai opté pour GitHub comme solution de développement collaboratif.

### 3. IDE : **Visual Studio Code**

Habitué à son interface, je me suis naturellement orienté vers cet IDE. Les nombreux add-ons disponibles pour ce logiciel ont également orienté mon choix

#### 4. Couche de compatibilité : **Windows Subsystem for Linux 2 (WSL 2)**

WSL2 permet d'exécuter des binaires Linux sous Windows et me permet d'avoir un environnement de développement constant entre mes deux PC

#### 5. Conteneurs : **Docker**

Afin d'avoir une cohérence entre mes deux environnements de développement, j'ai utilisé Docker et ses volumes afin d'avoir les mêmes applications sur mes deux machines.

En utilisant Docker Compose, j'ai créé plusieurs conteneurs :

- *MySQL* et *PhpMyAdmin* : SGBD SQL et son interface web
- *MongoDB* : SGBD NoSQL
- *MailDev* : simule une boîte e-mail
- *MinIo* : simule un conteneur AWS S3

J'ai également créé 3 volumes docker pour la conservation des données :

- *mysqldbdata* pour la base de données SQL
- *mongodbdata* pour la base de données NoSQL
- *minio-data* pour la sauvegarde des images

#### 6. Tests API : **Insomnia**

Afin de tester les différentes routes de mon API, j'ai préféré Insomnia à Postman qui est un logiciel similaire, mais je n'avais pas besoin de toutes les fonctionnalités que celui-ci propose. Insomnia me permet avec une interface simple d'effectuer mes tests et de sauvegarder les requêtes HTTP via un dépôt GIT, ce qui était suffisant pour mes besoins.

#### 7. Intelligence artificielle : **Chat GPT**

Difficile aujourd'hui de parler de projet informatique sans aborder la question de l'utilisation de l'intelligence artificielle.

Si au début j'ai été tenté de l'utiliser régulièrement, je me suis rapidement rendu compte que les réponses étaient peu adaptées à mes attentes.

Chat GPT a donc été utilisé dans ce projet uniquement pour m'aider à générer les jeux d'essais des bases de données, ainsi que dans de rares occasions afin de comprendre et corriger un bug.

## **Diagrammes UML**

### ***Diagramme de classe***

*(Annexe I – page 30)*

Afin d'organiser et comprendre les données avant de les implémenter dans une base de données, le MCD est un modèle visuel qui sert de plan pour décrire les données et leur relations sans se soucier de l'aspect technique.

Celui-ci avait été fourni avec le sujet du projet. Je m'en suis servi comme base afin d'établir mon diagramme de classes UML et ma base de données relationnelle, mais en y ajoutant des informations qui me paraissaient utiles pour une meilleure ergonomie du site.

### ***Diagramme de cas d'utilisation***

*(Annexe II – page 31)*

Le diagramme de cas d'utilisation est un modèle visuel qui sert de plan pour décrire les classes, leurs attributs, méthodes et relations sans se soucier des détails de l'implémentation.

Il permet de représenter les objets et leurs interactions de manière claire et organisée, facilitant ainsi la conception et la communication entre les membres de l'équipe de développement.

En me basant sur le cahier des charges, j'ai donc modélisé ce dernier.

### ***Diagrammes de séquence***

*(Annexe III – page 31 et 32)*

Le diagramme de séquence permet de représenter l'ordre des messages échangés entre les composants du système de manière claire et organisée, en suivant une « ligne de vie » temporelle.

Pour modéliser et comprendre le comportement que devra avoir mon application avant de la programmer, j'ai établi des diagrammes de séquence représentant différents parcours utilisateurs.

## Déploiement de l'application

### Front-end

- **Netlify**

Netlify est une plateforme d'hébergement d'applications web, adaptée aux projets Angular.

Elle autorise notamment de créer un pipeline CD basé sur la branche main, ce qui permet d'automatiser le déploiement de la branche « master » distante de mon projet, dès que la branche connaît des modifications.

### Back-end

- **Serveur**

J'ai choisi Heroku, une plateforme PaaS (Platform as a Service) ayant un haut niveau de sécurité qui permet de déployer rapidement des applications web. Les aspects majeurs qui ont guidé mon choix sont sa simplicité d'utilisation et Heroku CLI qui permet d'utiliser un pipeline CD pour déployer mes mises à jour du serveur.

- **Base de données MySQL**

Heroku permet l'utilisation d'add-ons, et JawsDB MySQL est le plus populaire pour gérer les bases MySQL, c'est donc vers ce dernier que s'est porté mon choix. Bien que l'accès par le terminal à JawsDB soit possible, il m'est apparu plus pratique d'utiliser un GUI pour travailler sur la base en distanciel. J'ai donc travaillé avec HeidiSQL qui est un outil de gestion de BDD gratuit open-source proposant des fonctionnalités amplement suffisantes pour ce projet.

- **Base de données MongoDB**

Il n'y a pas d'add-on dédié à MongoDB sur Heroku, mais en passant par Mongo Atlas, j'ai pu créer une BDD NoSQL accessible en ligne.

Ensuite, pour gérer cette base, j'ai utilisé le GUI fourni par Mongo, appelé Compass.

## Étapes du déploiement

### Back-end

J'ai commencé par déployer ma partie back-end. J'ai tout d'abord créé un compte sur Heroku, puis j'ai créé une nouvelle app. J'ai laissé Heroku choisir son nom pour moi, bien qu'il soit possible de le personnaliser.

J'ai ensuite intégré l'add-on JawsDB MySQL pour intégrer la base de données SQL à mon application.

Il a fallu ensuite renseigner, au moyen de l'interface de l'application, toutes les variables d'environnement nécessaires.



J'ai ensuite préparé mon application pour la production en renseignant notamment dans le fichier package.json la présence et la bonne configuration de la commande « npm start »

```
1 "scripts": {
2   "start": "NODE_ENV=production node server.js",
3   "dev": "nodemon -r dotenv/config ./server.js dotenv_config_path=../.env",
4   "test": "echo \"Error: no test specified\" && exit 1"
5 },
```

Je me suis ensuite connecté à Heroku par le terminal. Comme le front-end et le back-end de mon projet avaient un git commun, j'ai dû passer par une branche temporaire afin de n'uploader sur Heroku que la partie serveur de mon code :

```
thomas@BEANSIDHE:~/src/projects/arcadia$ git subtree split --prefix=server -b temp-branch
Created branch 'temp-branch'
d2311c54cf26d972238b3f08f3d73d6cada54b9b
thomas@BEANSIDHE:~/src/projects/arcadia$ git push heroku temp-branch:master
```

J'ai ensuite intégré mon jeu d'essai à la base de données MySQL avec l'application HeidiSQL, puis j'ai ensuite testé mes routes déployées avec Insomnia.

Concernant la partie NoSQL, j'ai créé un cluster Mongo Atlas et ai simplement renseigné les variables d'environnement de connexion dans Heroku, en adaptant les droits de l'api pour qu'elle n'ait accès qu'à la collection nécessaire.

## Front-end

Le déploiement du front-end a été effectué sur Netlify. Bien que cette solution propose également une configuration par CLI, il aurait été nécessaire que je me familiarise avec des fichiers de configuration, notamment pour le déploiement automatique et la gestion des variables d'environnement, tâche qui aurait été trop chronophage dans le délai imparti. J'ai donc choisi de passer par l'interface en ligne du site, qui propose des options suffisantes pour mon projet, et de tenter d'utiliser la solution par ligne de commande pour d'autres applications.

J'ai donc créé un compte Netlify et ai lié ce dernier à mon dépôt GitHub.

L'interface propose une partie « Build & Deploy » où je pouvais configurer les informations nécessaires au déploiement de mon front-end. La possibilité notamment de choisir le « Base directory » était particulièrement adaptée à ma configuration de travail.

**Build settings**

**Runtime:** Select ▾  
Runtimes are automatically installed on site creation. You can change this at any time, although it may affect your builds.

**Base directory:**   
The directory where Netlify installs dependencies and runs your build command.

**Package directory:**   
For monorepos, the directory that contains your site files, including the `netlify.toml`. Set this value only if it is different from the base directory.

**Build command:**

**Publish directory:**

**Functions directory:**   
The directory where Netlify can find your compiled functions to deploy them. Defaults to `netlify/functions` if not set. You can also define and override this setting in your site's `netlify.toml` file.

Extrait de la page de configuration de Netlify

Une fois mon application uploadée et déployée, j'ai vérifié directement en ligne son bon fonctionnement.



# Gestion du projet

## Diagrammes

J'ai commencé par cette étape afin d'avoir une vue précise des éléments qu'il faudrait développer pour répondre au cahier des charges. Les réalisations de ce bloc de travail sont décrites dans la documentation technique

## Recherche graphique

### Wireframes

*(Annexe IV)*

Afin d'avoir une idée générale de l'organisation visuelle pour l'utilisateur du site et l'ergonomie de ce dernier, j'ai établi des wireframes avec le logiciel Figma. Ces derniers m'ont permis de définir les différents éléments que comporteraient mes pages ainsi qu'une première idée des règles CSS à mettre en place pour que l'affichage soit réactif.

Je les ai créés avec une approche « mobile first », en effet la grande majorité des consultations de sites se font par des appareils mobiles (smartphones et tablettes), il est donc nécessaire aujourd'hui de proposer des interfaces adaptées prioritairement à ces outils.

### Charte graphique

*(Annexe V)*

Concernant la charte graphique, il fallait transmettre l'idée écologique formulée par le commanditaire du site. J'ai donc choisi un jeu de couleurs représentant la nature :

- Vert forêt
- Jaune soleil
- Bleu océan
- Beige sable

Pour les fontes, j'ai utilisé Google Fonts, qui permet une intégration sélective des fontes, de leur graisse et de leurs styles afin de ne pas surcharger l'application avec des données inutiles.

J'ai choisi Noto Sans pour les textes, car celle-ci est reconnue pour sa lisibilité, et Bricolage Grotesque pour les en-têtes, celle-ci étant suffisamment lisible en grand format.

J'ai également créé un logo sur Adobe Photoshop afin de donner au site une identité visuelle.

## Mockups

(Annexe VI)

Une fois ma charte graphique réalisée, j'ai pu réaliser les mockups, toujours avec le logiciel Figma. Cela m'a permis de confirmer le design du site et son ergonomie UI et UX.

## Préparation du développement

Git, Docker et NodeJS étaient déjà installés sur ma machine, je n'ai eu qu'à faire une mise à jour de chacun de ces programmes.

J'ai ensuite créé un dossier de travail et initialisé mon dépôt Git ainsi que « npm », le gestionnaire de packages pour NodeJS, dans ce dernier, avec les commandes :

```
git init et npm init
```

J'ai ensuite créé deux sous-dossiers, « client » et « server » pour séparer mes parties front-end et back-end.

J'ai ensuite configuré un fichier « docker-compose.yaml » afin de lancer les différents services nécessaires qui ont été présentés plus haut, en utilisant des variables d'environnement.

Depuis la version 26 du Docker Engine, l'indication de la version pour Docker Compose est dépassée.

```
1 services:
2   mysql:
3     image: mysql:8.4.0
4     container_name: mysqlldb
5     command:
6       - --mysql-native-password=ON
7       - --log-bin-trust-function-creators=1
8     environment:
9       MYSQL_ROOT_PASSWORD: ${BDD_ROOT_PASSWORD}
10      MYSQL_DATABASE: ${BDD_NAME}
11      MYSQL_USER: ${BDD_USER}
12      MYSQL_PASSWORD: ${BDD_PASSWORD}
13     ports:
14       - "3306:3306"
15     volumes:
16       - mysqlldbdata:/var/lib/mysql
17     networks:
18       - arcadia_network
```

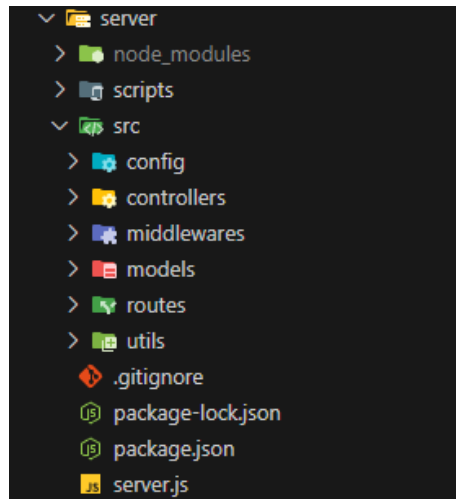
Extrait du fichier docker-compose.yaml.

## Back-end

NodeJS se base sur des middlewares pour la gestion des requêtes HTTP (req), le traitement des réponses (res) et la manipulation des données. Ces fonctions permettent d'ajouter des fonctionnalités adaptatives.

J'ai initialisé npm dans le sous dossier « server » afin d'avoir une gestion indépendante des packages NodeJS pour mon serveur.

J'ai organisé mes sous-dossiers afin d'avoir des tâches claires pour chaque entité qui seraient incluses dans ces derniers :



Arborescence du serveur

Pour des raisons pratiques, afin de ne pas installer inutilement des packages à la racine du projet, les scripts d'initialisation du serveur ont été inclus dans son dossier dans le répertoire « scripts ».

J'ai créé des routes CRUD génériques pour mes entités, en effet il m'est rapidement apparu que les instructions étaient très similaires, donc j'ai effectué une refactorisation de ces dernières afin d'obtenir un fichier « crud.js » qui gérerait les opérations simples, et pour les cas particuliers je les ai traitées au cas par cas dans leurs routes respectives.

Bien que l'ORM Sequelize permette de traiter les données sans passer par des commandes SQL, afin de respecter les indications pour cet ECF, toutes les requêtes ont été écrites dans ce langage.

```
1  const keys = Object.keys(recordData);
2  const values = keys.map((key) => recordData[key]);
3  const sql = `INSERT INTO ${tableName} (${keys.join(', ')}) VALUES (${keys
4    .map(() => '?')
5    .join(', ')})`;
6
7  const result = await sequelize.query(sql, {
8    replacements: values,
9    type: sequelize.QueryTypes.INSERT,
10 });
```

Exemple de requête SQL INSERT générique avec l'ORM Sequelize.

```

1 module.exports = (app) => {
2   app.post('/api/species/', authenticate(['ROLE_ADMIN']), crud.create);
3   app.get('/api/species/', crud.readAll);
4   app.get('/api/species/:id', crud.readById);
5   app.put('/api/species/:id', authenticate(['ROLE_ADMIN']), crud.update);
6   app.delete('/api/species/:id', authenticate(['ROLE_ADMIN']), crud.delete);
7
8   // Specie specific routes
9
10  app.get('/api/species/:id/animals', async (req, res) => {
11    try {
12      const specieId = req.params.id;
13      const sql = `
14        SELECT animalId, animalName, animalDescr, animalBirth, animalGender
15        FROM Animals
16        WHERE specieKey = ?`;
17      const [animals] = await sequelize.query(sql, {
18        replacements: [specieId],
19      });
20
21      if (animals.length === 0) {
22        return res.status(404).json({ error: 'No animals found for this specie' });
23      }
24
25      res.status(200).json(animals);
26    } catch (error) {
27      res.status(500).json({ error: error.message });
28    }
29  });
30 };

```

Exemple d'utilisation du CRUD générique avec route supplémentaire personnalisée

J'ai ensuite validé mes routes avec Insomnia. Ayant un CRUD générique, je n'ai pas eu à toutes les tester, mais les routes particulières ont été éprouvées individuellement afin de prévenir tout comportement inattendu.

## Front-end

Après avoir initialisé npm dans le dossier « client », j'y ai installé Angular CLI. Cette dernière permet de générer automatiquement de nombreux éléments du projet.

J'ai majoritairement utilisé :

- `ng generate component <composant>` pour créer le squelette d'un nouveau composant Angular, composé d'un fichier composant en lui-même, un fichier de template HTML, un fichier de styles dédié au template et un fichier pour les tests unitaires.
- `ng generate service <service>` pour créer des services, qui sont des classes injectables dans les différents composants pour centraliser notamment les appels API dans le cadre de cette application.

## Angular 17

Angular est un framework évoluant rapidement. Par rapport aux projets des versions précédant la 17, on remarque deux évolutions majeures :

- L'abandon de la standardisation des modules (NgModules) au profit des composants autonomes (Standalone Components)
- L'introduction des directives @if, @for, @switch en remplacement des directives \*ngIf, \*ngFor, et ngSwitch, les premières étant plus optimisées et ne nécessitent pas le CommonModule d'Angular.

Angular propose un moteur de template appelé Angular Template Syntax qui permet de lier les données et la logique des composants aux vues.

J'ai utilisé plusieurs fonctionnalités de ce moteur dans mon projet, en voici les principales :

- **Interpolation** : permet d'appeler directement la valeur d'une variable dans le template : `<p>{{ maVariable }}</p>`
- **Property binding** : permet de lier une propriété HTML avec une variable : `<img [src]="imageUrl">`
- **Event binding** : permet de lier un événement avec une méthode du composant : `<button (click)="maFonction()">Cliquez ici</button>`
- **Control flow** : permet d'afficher, de masquer et de répéter des éléments de manière conditionnelle :

```
@if (a > b) {  
  {{a}} is greater than {{b}}  
}
```

## Routage

Afin d'optimiser mon application, j'ai choisi d'utiliser la possibilité de lazy-loading, qui permet d'appeler les vues à la demande, en évitant de charger l'intégralité du site à son ouverture.

J'ai également distingué les routes du back-office et du front proprement dit, afin que mon routage soit plus simple à maintenir.

```

1  export const routes: Routes = [
2    {
3      path: '',
4      component: LandingClientComponent,
5      loadChildren: () => import('./client.routes').then((m) => m.default),
6    },
7    {
8      path: 'dashboard',
9      component: DashboardMainComponent,
10     loadChildren: () => import('./dashboard.routes').then((m) => m.default),
11   },
12   {
13     path: '404',
14     title: 'Page introuvable | Zoo Arcadia',
15     component: NotFoundComponent,
16   },
17   {
18     path: '**',
19     redirectTo: '404',
20   },
21 ];
22

```

Routage principal de l'application avec lazy loading des composants

## Back-office

J'ai commencé par coder le back-office du site. N'ayant pas trouvé de solution automatique d'intégration de tableau de bord, je l'ai développé intégralement.

Là encore, la répétition du code pouvant devenir rapidement importante, j'ai refactorisé mes composants afin de l'éviter au maximum. J'ai notamment créé un tableau et un formulaire configurables répondant à mes besoins.

The screenshot displays the 'Arcadia' back-office interface. On the left is a sidebar menu with options: Accueil, Horaires, Services, Habitats, Espèces, Animaux, Avis, and Utilisateurs. The main content area has a green header with 'Arcadia' and a user icon. Below the header is the title 'Especies' in large blue font. A filter bar shows 'Filtre Savane'. A table lists species with columns: Nom, Taxonomie, Description, Habitat, and Actions. The table contains four rows: Girafe, Zèbre, Gnou, and Antilope Cobe de Lechwie. Below the table is a pagination bar showing 'Items per page: 10' and '1 - 4 of 4'. A yellow circular button with a plus sign is next to the table. Below the table is a section titled 'Ajouter un enregistrement' (Add a record) with a form containing fields for 'Nom\*', 'Taxonomie\*', 'Description\*', and 'Habitat\*'. At the bottom of the form are two buttons: 'Enregistrer' (Save) and 'Annuler' (Cancel).

Exemple de vue du back office

```

1  export interface SqlViewDataConfig<T> {
2    label: string;
3    data: Observable<T[]>;
4    primaryKey: string | number;
5    displayColumns?: {
6      key: string;
7      label: string;
8    }[];
9    actions?: {
10     view?: boolean;
11     edit?: boolean | ((item: T) => boolean);
12     delete?: boolean | ((item: T) => boolean);
13     newSub?: boolean | ((item: T) => boolean);
14   };
15   booleanColumns?: string[];
16   sortable?: boolean;
17   formFields?: FormField[];
18   customValidators?: ValidatorFn[];
19   noFilter?: boolean;
20   noPaginator?: boolean;
21   imageManager?: ImageManager;
22 }

```

Interface Angular de configuration des vues du back-office

Pour les filtres et la pagination, la bibliothèque Angular Material propose ces fonctionnalités nativement, l'intégration de ces features demandées par le client ont donc été simplifiées.

## Vues visiteur

Pour la partie visiteur, j'ai utilisé le framework Bulma pour la mise en page. Il fournit un ensemble d'outils permettant de composer une interface réactive.

J'ai aussi intégré Angular Material qui propose des composants UI adaptés à Angular et respectant le système de conception Material Design de Google, l'un des plus populaires dans la conception d'UI.

L'intégration de carrousels m'a été assez difficile avec ces deux seules bibliothèques ainsi que mes connaissances associées, j'ai donc dû intégrer sélectivement le carrousel de Bootstrap pour les réaliser.

J'ai suivi les différents mockups que j'avais créés, en y ajoutant quelques éléments réactifs, notamment la barre de menu supérieure afin d'apporter du dynamisme à mon application



Barre de menu avant défilement



Barre de menu après défilement



```

1  scrolled = false;
2  @HostListener('window:scroll', ['$event'])
3  onWindowScroll() {
4    this.scrolled = window.scrollY > 100;
5  }

```

Contrôle du comportement de la barre de menu

Là encore le moteur de template m'a été utile afin de charger mes données dynamiquement.

J'initialise mes données dans le composant en faisant appel au service concerné :

```

1  getReports(): void {
2    this.reportService.getAnimalReports(this.animal.animalId).subscribe({
3      next: (reports) => {
4        if (reports.length > 0) {
5          // Utiliser reduce pour trouver le rapport le plus récent
6          const lastReport = reports.reduce((latest, report) => {
7            return new Date(report.reportDate) > new Date(latest.reportDate) ? report : latest;
8          }, reports[0]);
9
10         // Mettre à jour this.animal.reports avec le rapport le plus récent
11         this.animal.reports = [lastReport];
12       } else {
13         console.log('Aucun rapport trouvé pour cet animal.');

```

Récupération du dernier rapport vétérinaire pour un animal

Puis j'utilise ces données dans mon template HTML:

```

1  @if(animal.reports && animal.reports.length > 0){
2    <div class="vet-report p-4">
3      <h5><strong>Rapport du vétérinaire: </strong>{{ animal.reports[0].reportState }}</h5>
4      <p>
5        <strong>Dernière visite:</strong> {{ animal.reports[0].reportDate | date : 'dd/MM/yyyy' }}
6      </p>
7      <p>{{ animal.reports[0].reportDetails }}</p>
8    </div>
9  }

```

Intégration dans le template des données avec utilisation à la ligne 5 d'un pipe Angular



## Sécurité

Dans le cadre du développement d'une application web, la question de la mise en place de procédures de sécurisation de l'app afin d'éviter les attaques malveillantes qui pourraient conduire notamment à des vols de données, perturbations du service, pertes financières, atteintes à la réputation et à des conséquences légales, est essentielle.

Il est donc nécessaire de mettre en place des procédures de protection de l'application, et d'effectuer une veille de sécurité régulière.

### Sécurité de l'application

Afin d'éviter les failles de sécurité dans les modules npm, il faut que ces derniers soient mis régulièrement à jour.

La mise à jour se fait avec une commande chaînée :

```
npx npm-check-updates -u && npm install
```

Celle ci, dans le cadre de mon projet, doit être exécutée à 3 endroits : racine du projet, dossier « client » et dossier « server »

### Sécurité back-end

Concernant mon back-end, j'ai mis en place plusieurs sécurités :

- **CORS** : le middleware CORS permet de gérer les origines autorisées et restreindre les requêtes venant de domaines non autorisés.
- **Express-rate-limit** : ce middleware limite le nombre de requêtes provenant de la même IP pour empêcher les attaques par déni de service (DoS).
- **Json Web Token** : lors de la connexion d'un utilisateur, un jeton JWT est généré. Celui-ci permet la transmission sécurisée d'informations entre deux parties. Je l'ai utilisé pour authentifier les utilisateurs dans mon app, et vérifier leurs droits d'accès à certaines routes.

Il est stocké dans la machine de l'utilisateur sous forme de cookie.

```
1 const token = jwt.sign({ userId: user.userId, userRole: user.userRole }, privateKey, {  
2   expiresIn: '24h',  
3 });
```

Création du token JWT

- **Protection des routes** : grâce à un middleware d'authentification, les routes protégées ne sont accessibles qu'aux utilisateurs ayant un rôle accepté

```
1 module.exports = (app) => {  
2   app.post('/api/services/', authenticate(['ROLE_ADMIN', 'ROLE_EMPLOYEE']), crud.create);  
3   app.get('/api/services/', crud.readAll);  
4   app.get('/api/services/:id', crud.readById);  
5   app.put('/api/services/:id', authenticate(['ROLE_ADMIN', 'ROLE_EMPLOYEE']), crud.update);  
6   app.delete('/api/services/:id', authenticate(['ROLE_ADMIN', 'ROLE_EMPLOYEE']), crud.delete);  
7 };
```

- **Validation des entrées** : Sequelize nécessite l'implémentation de modèles qui peuvent être ensuite validés avant d'exécuter la requête.

```

1  userEmail: {
2    type: DataTypes.STRING,
3    allowNull: false,
4    unique: {
5      msg: 'Cette adresse email est déjà utilisée',
6    },
7    validate: {
8      isEmail: {
9        msg: 'Doit être une adresse email valide.',
10     },
11     notEmpty: {
12       msg: "L'adresse email ne peut pas être une chaîne vide",
13     },
14     notNull: {
15       msg: "L'adresse email est une propriété requise",
16     },
17   },
18 },

```

Extrait d'un modèle Sequelize qui vérifie si l'entrée est de type STRING, unique dans la BDD, de format email, une chaîne non vide, et non nulle

J'ai de plus intégré la validation du format d'id entré dans les requêtes avec des RegEx :

```

1  function isValidId(id) {
2    const isNumericId = /^\d+$/i.test(id);
3    const isUuid = /^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$/i.test(id);
4    return isNumericId || isUuid;
5  }

```

- **Injections SQL** : l'injection de code SQL est empêchée par les prepared statements dans mes requêtes, en remplaçant la valeur de la requête par un '?'

```

1  let fetchSql =
2    getReadByIdQuery(modelName, lastId) ||
3    `SELECT * FROM ${tableName} WHERE ${primaryKeyField} = ?`;
4
5  const [lastRecord] = await sequelize.query(fetchSql, {
6    replacements: [lastId],
7    type: sequelize.QueryTypes.SELECT,
8  });
9

```

## Sécurité front-end

Angular implémente automatiquement certaines sécurités :

- **Sanitization** : nettoie automatiquement les données avant de les insérer dans le DOM pour prévenir les attaques de type Cross-Site Scripting (XSS). Par exemple, les liens, les styles et les contenus HTML dangereux sont automatiquement désinfectés.
- **Template Injection Prevention** : protège contre les injections de modèles en utilisant une syntaxe de liaison de données qui empêche l'exécution de code non désiré dans les templates.
- **AOT (Ahead-of-Time) Compilation** : La compilation AOT convertit les templates Angular et le code en JavaScript avant l'exécution, ce qui réduit les risques d'injection de code malveillant à l'exécution.
- **Strict Contextual Escaping (SCE)** : Angular utilise le SCE pour vérifier les valeurs utilisées dans des contextes sensibles, tels que les URL et les scripts, afin de s'assurer qu'ils sont sûrs.

J'ai intégré des sécurités supplémentaires :

- **Json Web Token** : Il faut également implémenter le JWT côté client afin que les requêtes soient authentifiées par le serveur. Pour cela, il faut l'intégrer aux headers. Pour éviter la répétition du code, j'ai créé un intercepteur qui l'ajoute automatiquement, avec l'en-tête « Authorization »

```
1 export function AuthInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn) {
2   const authService = inject(AuthService);
3   const userToken = authService.currentUserValue?.userToken;
4
5   if (userToken) {
6     let headers = req.headers.set('Authorization', `Bearer ${userToken}`);
7
8     // Vérifie si la requête est multipart/form-data pour éviter une erreur de payload
9     if (!(req.body instanceof FormData)) {
10      headers = headers.set('Content-Type', 'application/json');
11    }
12
13    const modifiedReq = req.clone({ headers });
14    return next(modifiedReq);
15  }
16
17  return next(req);
18 }
```

- **Router Guards** : Angular fournit des guards afin de protéger les routes. Je les ai utilisés dans mon back-office afin d'empêcher un utilisateur malveillant d'accéder à des données non autorisées.

```
1  {
2    path: 'service',
3    title: 'Services | Arcadia admin',
4    loadComponent: () =>
5      import('@app/components/admin/partials/service-admin/service-admin.component').then(
6        (m) => m.ServiceAdminComponent
7      ),
8    canActivate: [AuthGuard],
9    data: { roles: ['ROLE_ADMIN', 'ROLE_EMPLOYEE'] },
10  },
```

Protection d'une route avec un guard, autorisant uniquement les rôles désignés par la propriété « data »

## Veille sécuritaire

Dans le cadre du développement d'une application, la veille sécuritaire est importante car elle permet de connaître les dernières alertes de sécurité, de connaître les bonnes pratiques de programmation afin d'éviter les failles. J'ai consulté plusieurs sources tout au long du développement de ce projet.

- **Agence nationale de la sécurité des systèmes d'information (ANSSI)** : elle met notamment à disposition des guides de bonnes pratiques pour la sécurité des applications web et des réseaux.
- **CERT-FR** : recense et met à disposition les alertes, menaces et avis de sécurité. Son bulletin régulier permet d'avoir rapidement une vue d'ensemble sur les dernières menaces, et ainsi savoir si notre projet est directement ou indirectement concerné.
- **Angular.dev** : site officiel d'Angular, le chapitre de sa documentation 'Best Practices' contient un volet sécurité très bien documenté pour développer une application peu sensible aux attaques.
- **NodeJS – Security** : Le site de NodeJS a une page dédiée à sa sécurité, et renvoie vers deux sites (un groupe Google et un blog) répertoriant les dernières alertes de sécurité et corrections de failles.
- **Owasp.org** : Certainement la source la plus complète en termes de cybersécurité, son Top Ten met en avant les principales menaces auxquelles sont exposés les sites Internet.

## Recherche anglophone

J'ai effectué de nombreuses recherches sur Internet durant ce projet.

L'une d'entre elles a été de comprendre comment créer un service injectable avec le décorateur @Inject

La documentation d'Angular fournit un article détaillé pour cette action :

<https://angular.dev/guide/di/creating-injectable-service>

« Service is a broad category encompassing any value, function, or feature that an application needs. A service is typically a class with a narrow, well-defined purpose. A component is one type of class that can use DI.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related features from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component. You can also make your application more adaptable by configuring different providers of the same kind of service, as appropriate in different circumstances.

Angular does not enforce these principles. Angular helps you follow these principles by making it easy to factor your application logic into services and make those services available to components through DI. »

En voici la traduction :

Le service est une vaste catégorie qui englobe toute valeur, fonction ou caractéristique dont une application a besoin. Un service est généralement une classe dont l'objectif est étroit et bien défini. Un composant est un type de classe qui peut utiliser l'injection de dépendance.

Angular distingue les composants des services afin d'accroître la modularité et la ré-utilisabilité. En séparant les fonctionnalités liées à la vue d'un composant des autres types de traitement, vous pouvez rendre vos classes de composants légères et efficaces.

Idéalement, le rôle d'un composant est de faciliter l'expérience de l'utilisateur et rien de plus. Un composant doit présenter des propriétés et des méthodes pour la liaison de données, afin d'assurer la médiation entre la vue (rendue par le modèle) et la logique de l'application (qui inclut souvent une certaine notion de modèle).

Un composant peut déléguer certaines tâches à des services, comme l'extraction de données du serveur, la validation des entrées de l'utilisateur ou l'enregistrement direct dans la console. En définissant ces tâches de traitement dans une classe de service injectable, vous les mettez à la disposition de n'importe quel composant. Vous pouvez également rendre votre application plus adaptable en configurant différents fournisseurs du même type de service, selon les circonstances.

Angular n'applique pas ces principes. Angular vous aide à suivre ces principes en facilitant la factorisation de votre logique d'application en services et en rendant ces services disponibles aux composants à travers l'injection de dépendance.

## Conclusion

Concernant mon application, quelques bugs subsistent, notamment l'affichage des images dans le tableau de bord utilisateur qui ne se met pas à jour automatiquement. Ceci pourrait être corrigé par l'utilisation des « Angular signals », concept apparu avec Angular 16 mais ayant connu depuis de nombreuses évolutions, pour arriver à une version stable avec la version 18 d'Angular.

Une amélioration de l'UX et de l'UI visiteur serait également un plus, en effet une application pour un zoo devrait selon moi avoir une interface plus ludique.

Ayant de bonnes connaissances du framework Bootstrap, je pense avoir commis une erreur en utilisant Angular Material et Bulma. En effet j'ai dû également intégrer pendant le développement leurs syntaxes et particularités. Sans être une perte de temps pour mon apprentissage, j'aurai pu consacrer ce temps à améliorer d'autres aspects de l'application.

J'ai cependant réussi à mettre en place de nombreux composants réutilisables, notamment le CRUD générique pour la base MySQL de mon serveur, ainsi que les tableaux et formulaires paramétrables de mon tableau de bord.

De plus, l'expérience acquise lors de ce projet sera bénéfique pour ceux à venir.

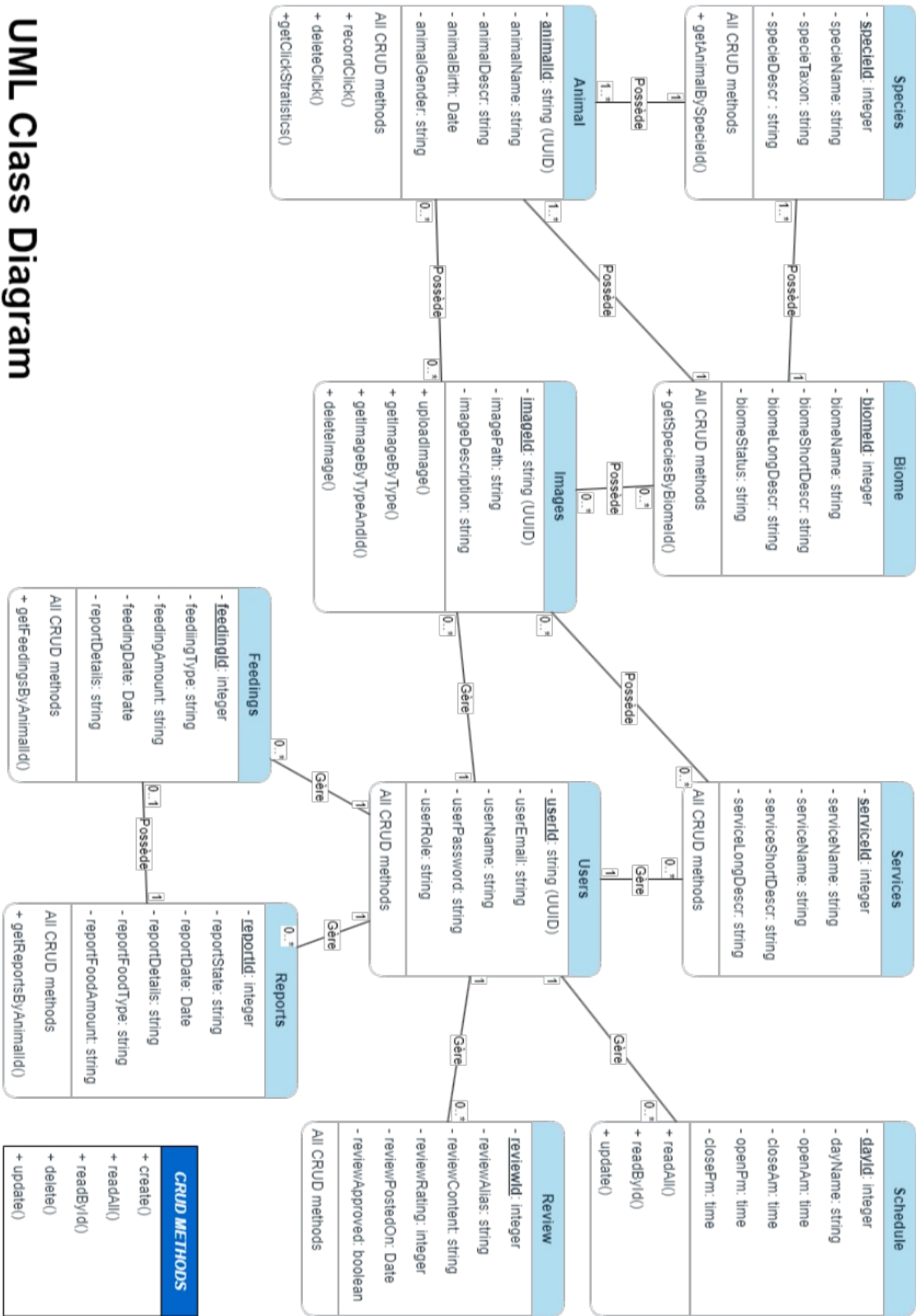
Ce projet m'a permis d'appréhender le travail d'un développeur, en mettant en pratique les notions acquises lors de ma formation, mais surtout combien il est important maintenir ses compétences à jour concernant la sécurité et les bonnes pratiques de programmation.

Prendre connaissance des documentations officielles, des évolutions des langages et des frameworks est un aspect tout aussi important du métier.

J'ai pu me confronter aux problèmes que rencontrent les concepteurs d'applications, notamment la résolution de bugs et les problèmes lors du déploiement.

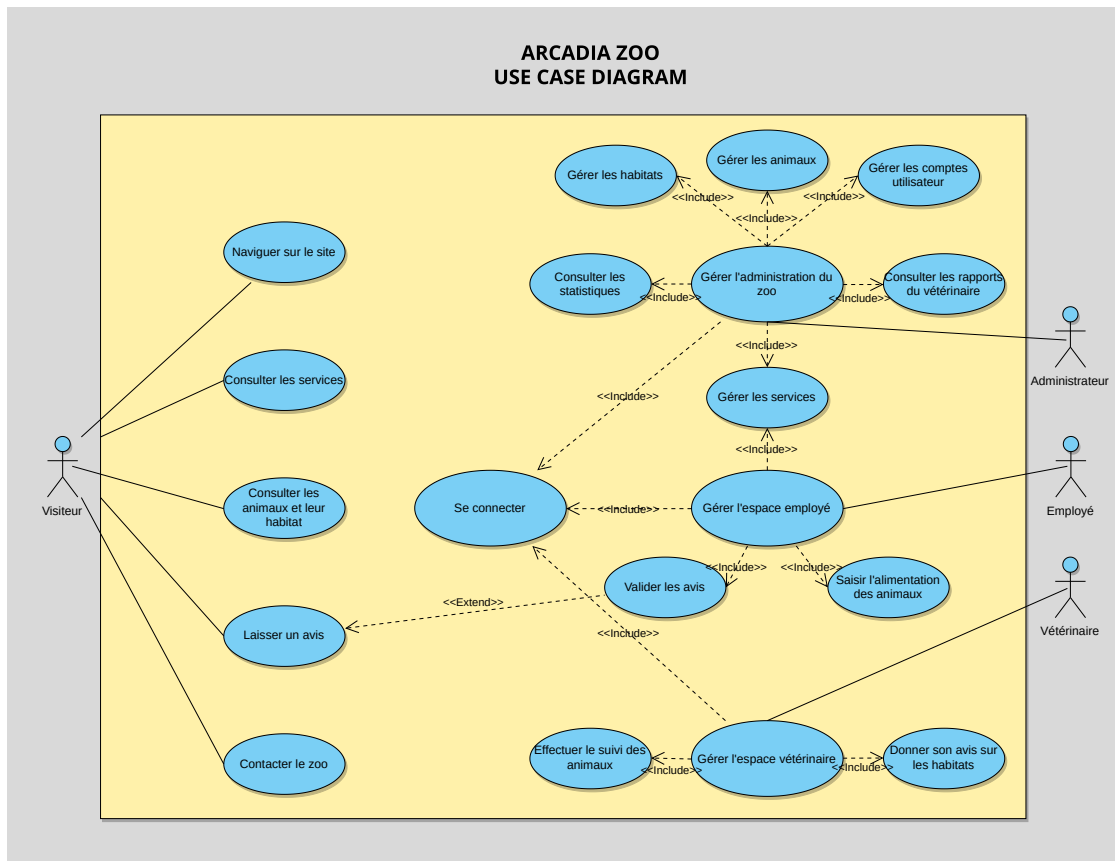
# Annexes

I. Diagramme de Classes



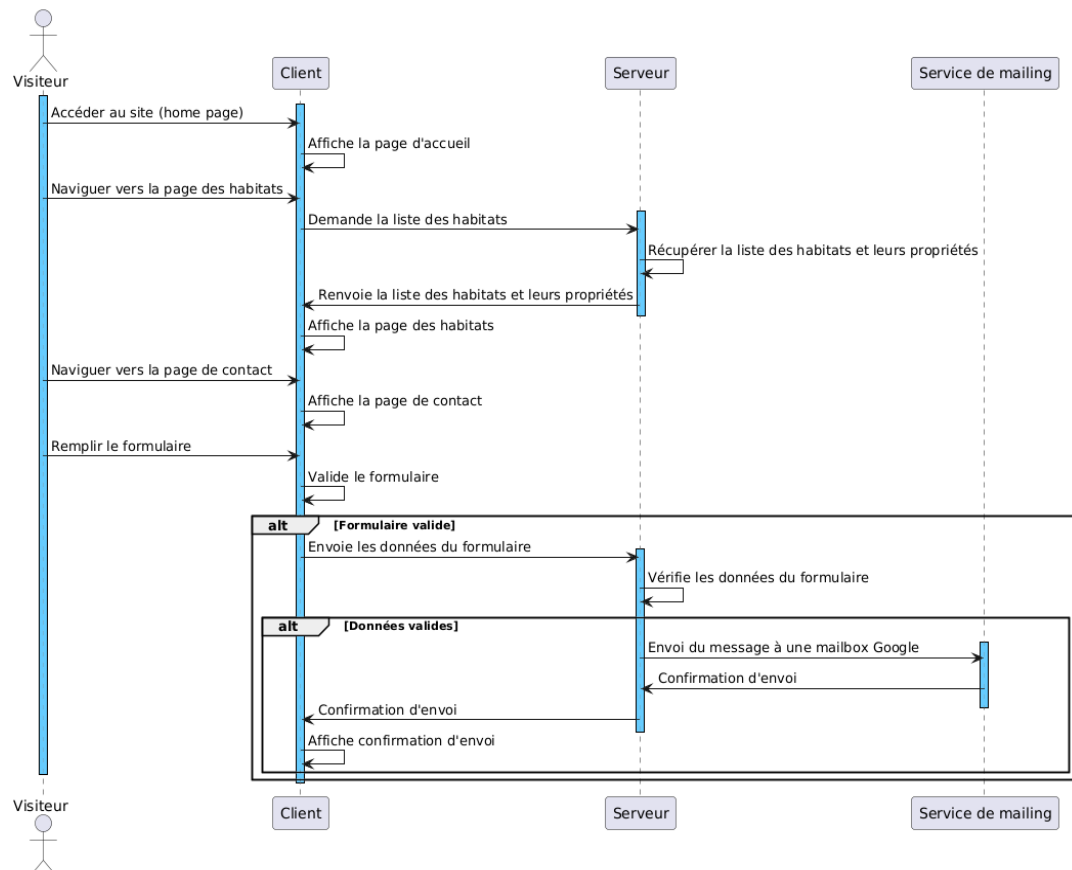


## II. Diagramme de Cas d'Utilisation

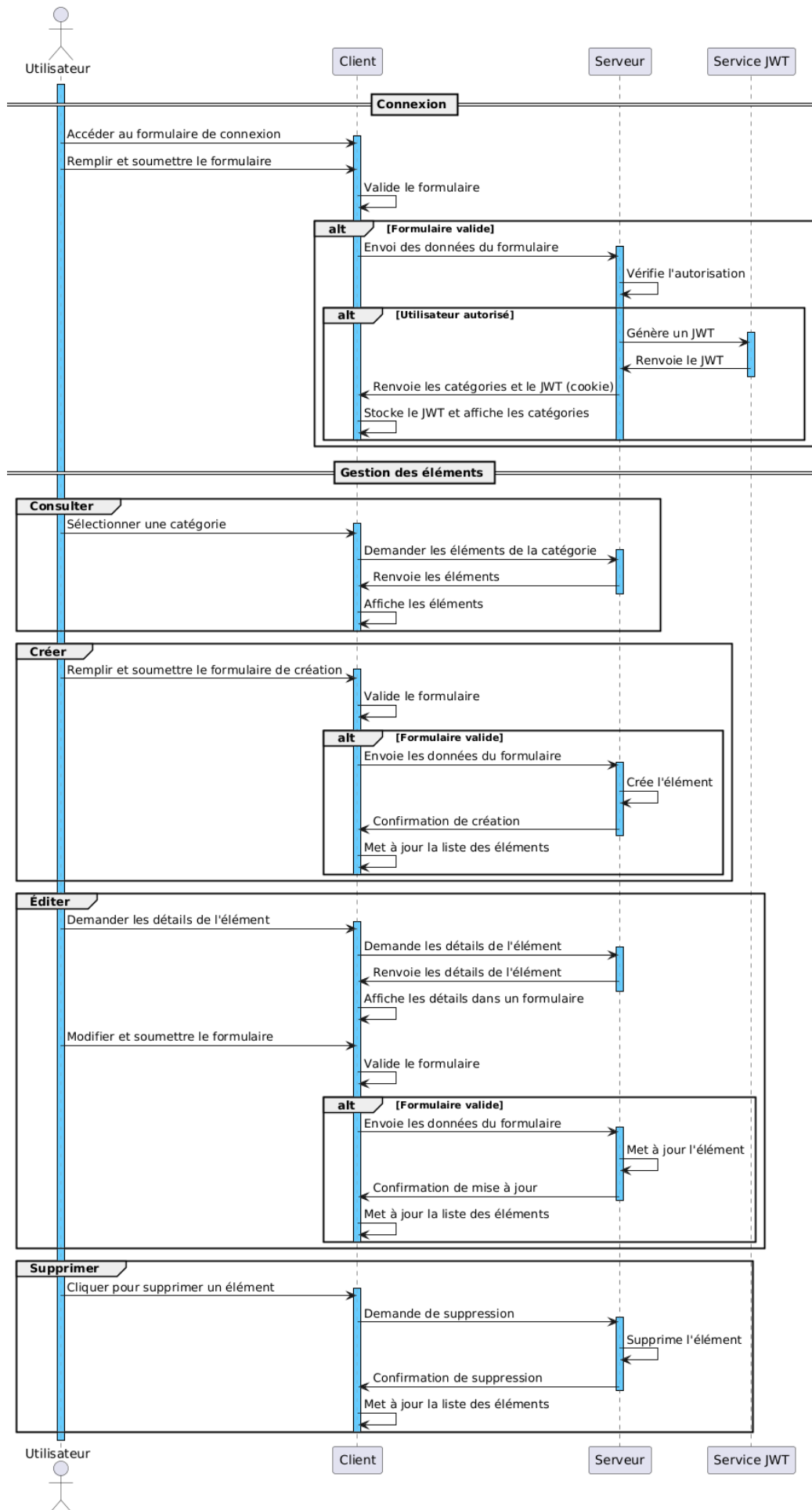


## III. Diagrammes de Séquences

### 1. Exemple de parcours visiteur



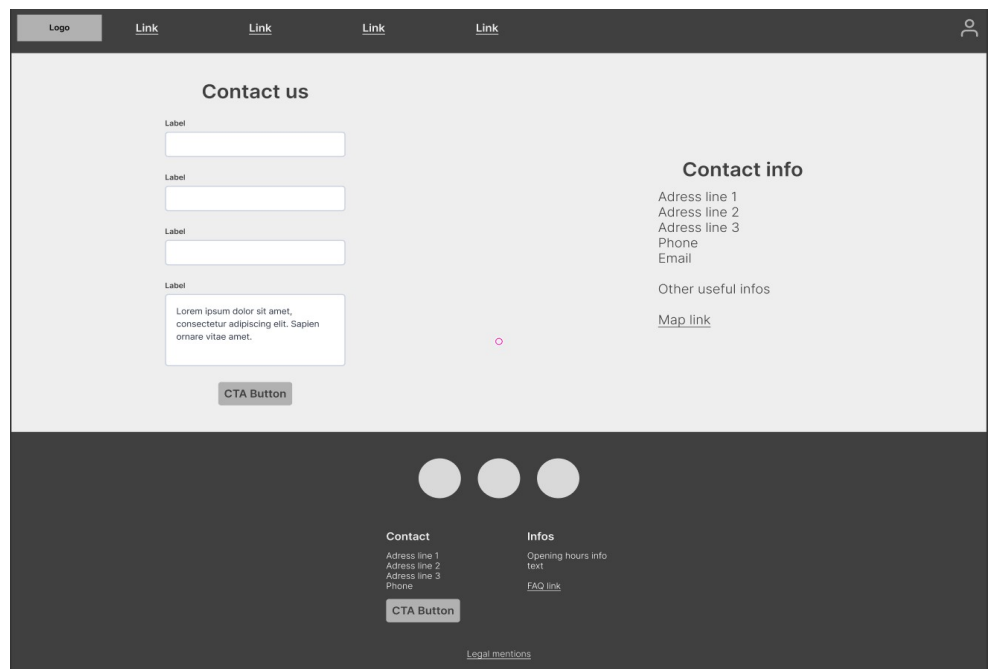
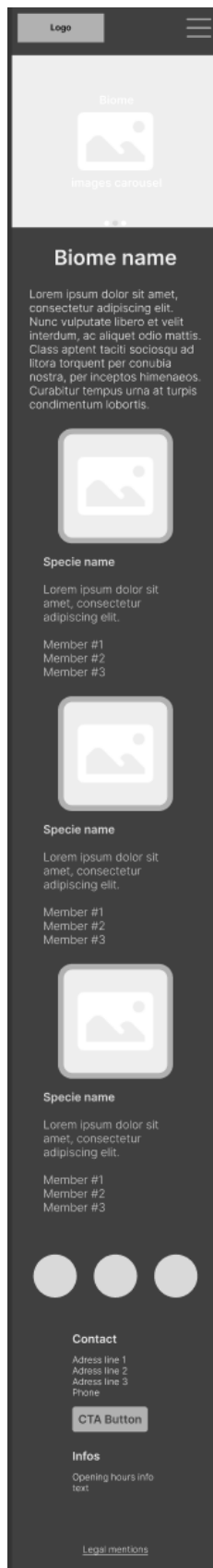
## II. Exemple de parcours utilisateur



## IV.Wireframes

Les wireframes ont été créés sous Figma. L'ensemble des vues sont disponibles à cette adresse : [Wireframes Figma](#)

### Exemples de Wireframes réalisés



## V. Charte Graphique

<b>Vert forêt</b> #0B6623	<b>Beige sable</b> #F4F1DE	<b>Jaune soleil</b> #FFC107	<b>Bleu océan</b> #0077B6
------------------------------	-------------------------------	--------------------------------	------------------------------

**Textes : Noto Sans**


Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quis ipsum suspendisse ultrices gravida. Risus commodo viverra mtaecenas accusan lacus vel facilisis.

**Headers : Bricolage Grotesque**

**Header 1**

**Header 2**

**Header 3**



**Menu**

**Headline**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Quis ipsum suspendisse ultrices gravida. Risus commodo viverra mtaecenas accusan lacus vel facilisis.

**Effectuer** **Annuler**

# VI. Mockups

L'ensemble des mockups est disponible en cliquant sur ce lien : [Mockups Figma](#)



Exemples de mockups réalisés avec Figma

