

Documentation technique

Zoo Arcadia

Sommaire

- 1. Présentation du projet**
- 2. Environnement de travail et stack technique**
- 3. Réflexion**
- 4. Recherche du besoin client**
- 5. Méthodologie**
- 6. Le maquettage**
- 7. Les diagrammes**
- 8. Le Frontend**
- 9. Le Backend**
- 10. Lier le frontend et le backend**
- 11. Le déploiement**
- 12. Fin du projet**

Sullivan KOWALSKI

2024/2025

1.Présentation du projet

Dans ce projet, nous allons concevoir une application web pour José, responsable du zoo Arcadia, afin d'augmenter sa visibilité et sa notoriété. Le thème du site portera sur l'écologie et la nature. Il souhaite que les visiteurs puissent avoir accès à tous les animaux, leur état, leurs habitats mais aussi, les services du zoo et les horaires. Il y aura également une interface utilisateur pour le personnel afin qu'il puisse faire des rapports vétérinaires sur les animaux, gérer les avis clients et gérer les habitats naturels du zoo.

2.Environnement de travail et stack technique

Pour la conception du site, je décide d'utiliser tout ce dont j'ai appris durant les cours. J'installe donc mon environnement de travail à savoir :

- Un ordinateur (Windows) et une connexion internet
- **Visual Studio Code**
- **Xampp**: Il me permettra de simuler un serveur en local afin de tester mon site et ses fonctionnalités en temps réel
- **PHP version 8.2.12**: Un langage de programmation qui me servira pour la partie Backend de du site
- **My SQL workbench 8.0**: Me permettra de créer ma base de données relationnelle
- **Mongo DB**: Me permettra de créer la base de données non relationnelle
- **Postman**: Me permettra de tester mes requêtes api
- **Figma**: Pour la création des maquettes avant la partie développement
- **Trello**: Pour la gestion et la planification du projet grâce à la méthode KANBAN
- **Always Data, PlatformSh**: Pour le déploiement des deux parties
- **Filezilla**: Pour les transferts de fichier en local vers un serveur distant
- **GIT,GITHUB**: Pour le versionnage de mon code
- **Langages**: HTML 5, CSS 3, JAVASCRIPT, PHP 8.2.12
- **Frameworks**: Symfony, PHPUnit

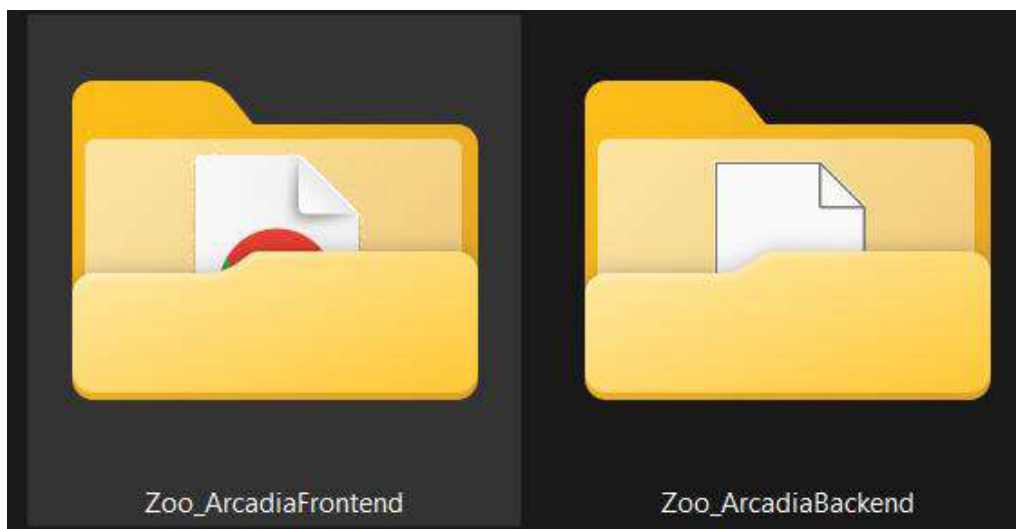
3.Réflexion

Une fois l'environnement de travail installé, je choisis les langages et les différents frameworks qui me serviront à travailler.

Pour la partie Frontend, je serais sur du **HTML** pour le langage de balisage, **CSS** pour la conception esthétique puis **JAVASCRIPT** pour le dynamisme et l'interactivité de mes pages. Il s'agit d'un langage polyvalent que j'apprécie particulièrement de par sa puissance mais aussi sa communauté active. J'ai fait le choix de ne pas utiliser de frameworks pour le frontend, pour la simple et bonne raison que je suis novice, et que la meilleure façon d'apprendre et de comprendre le développement, selon moi, était d'écrire le code de mon site, moi-même.

Pour la partie Backend, le langage **php** est de prime pour l'exécution côté serveur. Il intègre facilement le **html** et le **css**. Il allie **performance** et **sécurité**. Pour cette partie cette fois ci, j'utiliserai le frameworks **Symfony**. **Simple** et **efficace**, il suit le modèle **MVC** (Modèle, Vue, Contrôleur) et est conforme aux bonnes pratiques de développement avec des composants modulables et réutilisable, cela me semble idéal pour le projet.

Il y aura donc deux gros dossiers, **le frontend** et **le backend**, que je préfère séparés pour la lisibilité du projet et qui seront travaillés séparément. Chacun aura un repository distant sur **github** avec **3 branches**. La branche **main**, **development** et **fonctions** pour tester les fonctionnalités. La branche fonctions sera **mergée** sur la branche development une fois les tests de fonctionnalités OK, puis la branche development, **mergée** sur la branche principale main une fois que tout le projet de développement sera OK.



Ces deux parties communiqueront via une **API RESTful**, localement dans un premier temps. Le frontend appellera le backend via la fonction **FETCH** qui permettra de faire des requêtes auprès de l'api.

4.Etape du besoin client

Une des premières étapes, des plus importantes est **l'identification des besoins** de notre client, et il est nécessaire pour cela, d'établir un **cahier des charges** regroupant toutes les notions importantes mais aussi les **étapes du processus** de développement de l'application.

Aujourd'hui, nous savons que José a besoin d'une application web Vitrine, qui sera à disposition pour les utilisateurs, dans le but **d'augmenter sa visibilité** mais aussi la **fréquentation du zoo**.

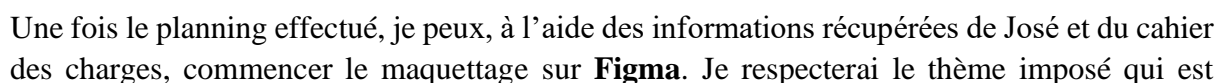
Il y aura une espace utilisateur **pour le personnel**, à savoir, **l'admin**, les **employés** et les **vétérinaires** où ils pourront se connecter avec leur email et leur mot de passe préalablement

Les visiteurs eux, pourront simplement visiter le site.

Une fois tout l’environnement installé et préparé. Il est important de **planifier** son travail, Et pour cela, j’utiliserai **Trello**, qui est un gestionnaire de projet où j’emploierai la méthode **Kanban**. (Voir le document “Gestion de projet”).

Pour chacune de ces colonnes, il y a, à l'intérieur des cartes avec des couleurs représentant chacune une fonctionnalité ou tâche. En fonction de l'avancement du travail, ces cartes seront **déplacées** dans une des colonnes citées ci-dessus.

Il est vrai aussi que l'on tombe parfois sur des imprévus ou des changements de dernières minutes dans le développement qui ne seront pas forcément notés dans le planning, cela dit, Trello est un très bon support et, est très efficace et me servira dans mes futurs projets.



l'écologie et proposerai des couleurs en adéquations avec celui-ci. (Ci-joint la charte graphique).

Dans un premier temps je déterminerai les différentes zones de la page d'accueil du site, une page clé pour le visiteur. En commençant par un **ZONING** suivit d'un **WIREFRAME**, un **PROTOTYPE**, puis un **mockup** en mobile first.

Ces maquettes seront présentées en format **bureau** et **mobile** afin de montrer le côté **responsive**, les fonctionnalités réalisables et les possibilités de développement.

Il y aura donc **3 pages** du site à présenter : la page d'accueil, la page de connexion et la page de contact sous tous ces formats.

J'ai opté pour un design **moderne**, à la hauteur de mes compétences bien entendu, avec des couleurs nous rappelant **la nature et l'écologie**.



7. Les diagrammes

Après avoir travaillé le maquettage, la partie visible du site, je m'attaque aux différents **diagrammes**. Ils me permettront de **schématiser mes idées** et de **réfléchir à l'architecture du système côté serveur**. Ils montreront les différentes **interactions entre les utilisateurs et le système** par exemple mais aussi **structurer les données, leur relation** entre elles et, décrira l'ordre des interactions entre les composants ou classes au fil du temps.

Je pars donc sur trois types de diagrammes :

- Diagramme d'utilisation** pour les interactions entre les utilisateurs et le système
- Diagramme de séquence** pour l'ordre des interactions dans un cas d'utilisation, dans le temps
- Diagramme de classe** pour définir les différentes entités et données dans le système et leur relation

Diagramme d'utilisation

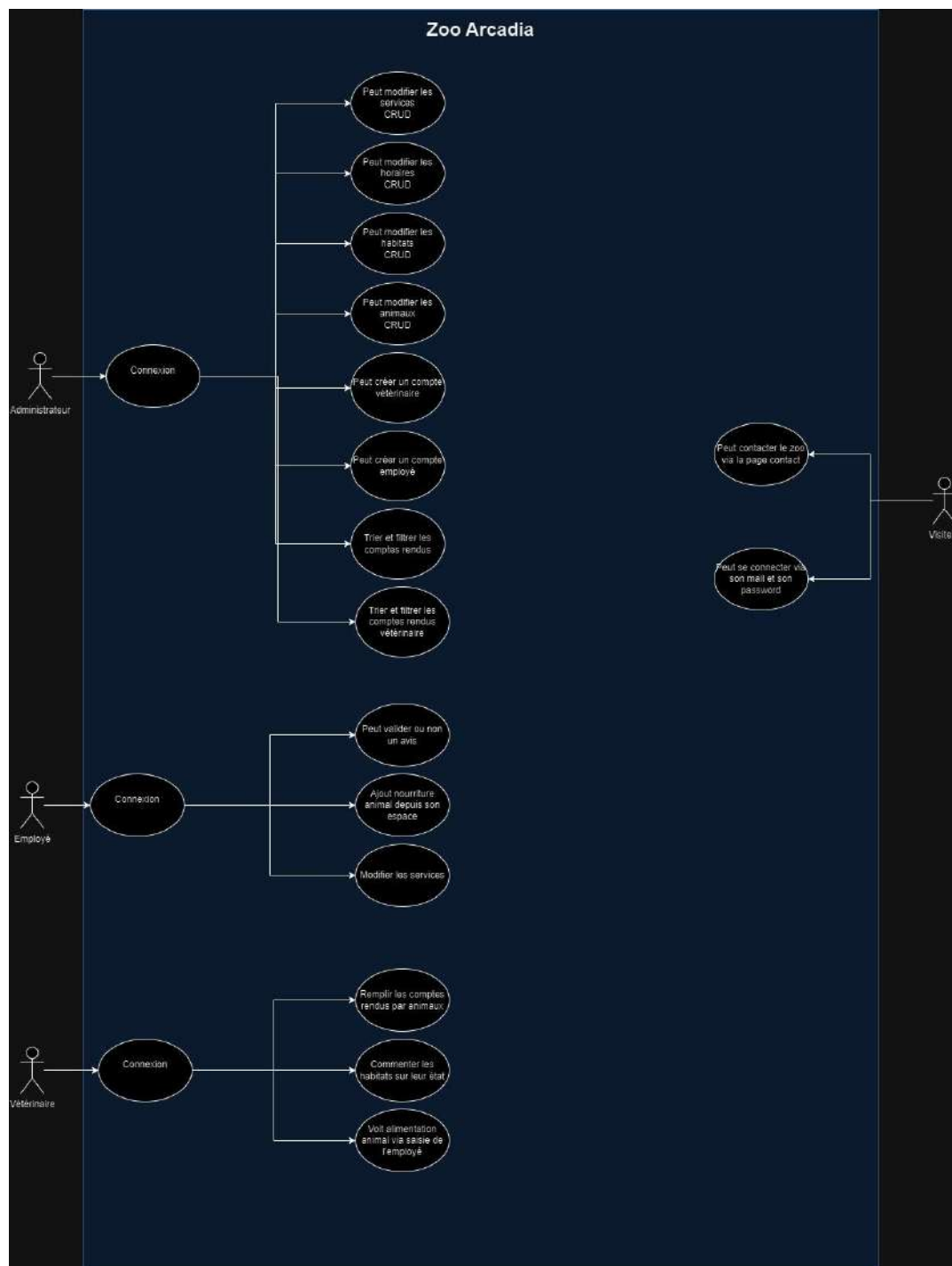


Diagramme de séquence ADMIN

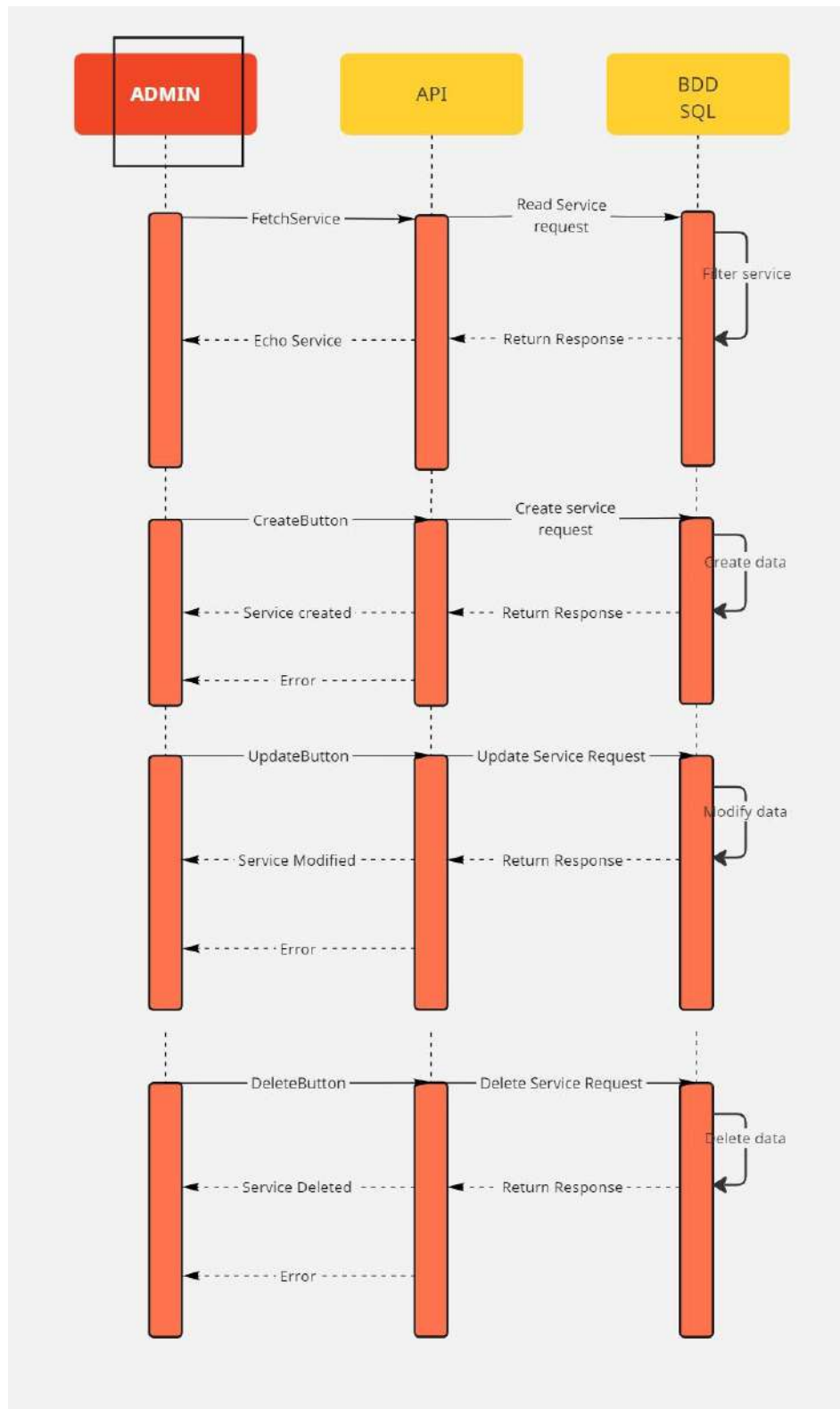
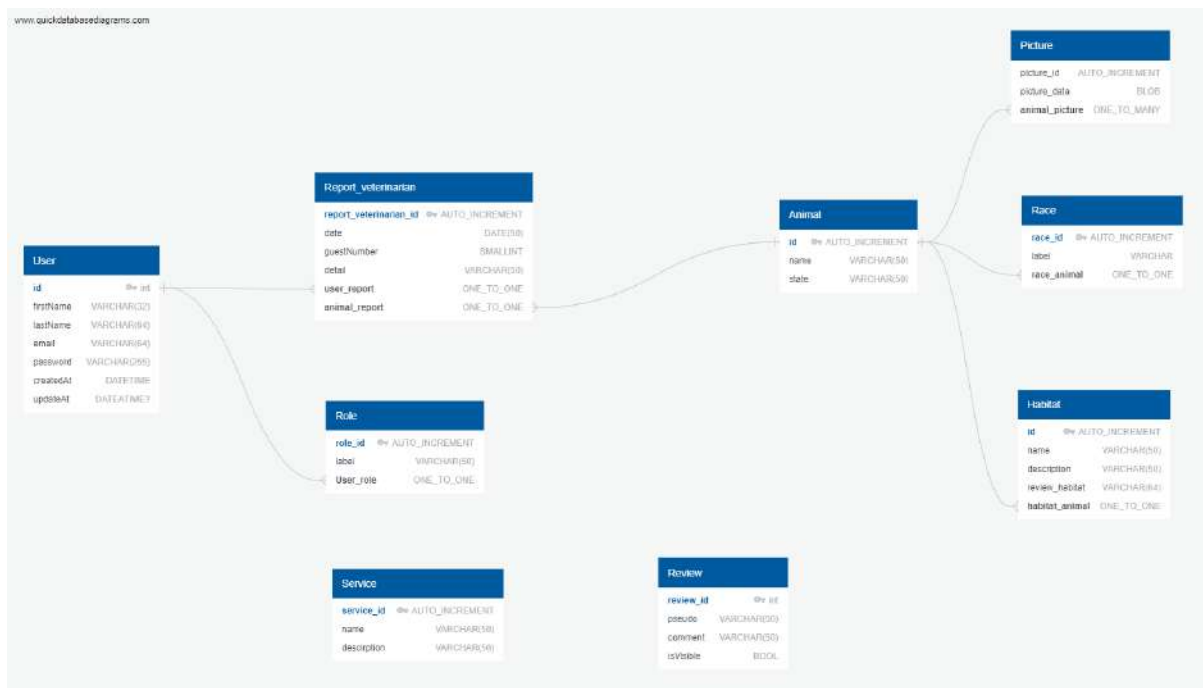


Diagramme de classe

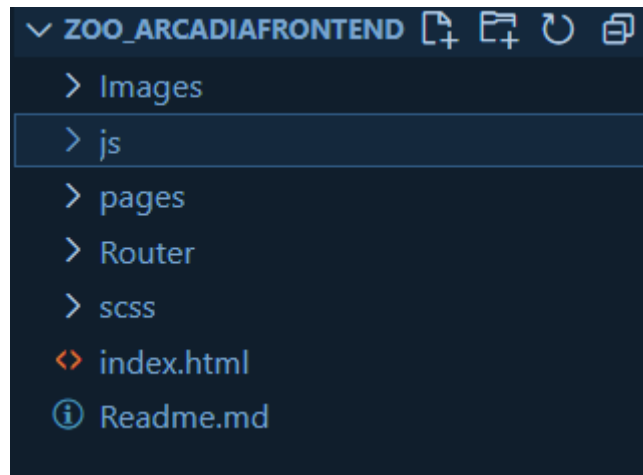


8.Le Frontend

Le maquettage étant fait, je peux attaquer le développement de l'application. En commençant tout logiquement par le **frontend**, qui est la partie **côté client** et visible. Je me concentre dans un premier temps sur la page d'accueil, **index.html**.

L'arborescence du dossier Frontend contiendra 5 sous-dossiers comme appris durant les cours :

- IMAGE** : Contient toutes les images du site
- JS** : Contient le script.js ainsi que les autres fichiers js.
- PAGES** : Contient toutes les pages du site
- ROUTER** : Contient les fichiers js avec le système de routage de l'application
- SCSS** : Contient le CSS de l'application
- INDEX.HTML** : le fichier source du projet
- README** : Contient les informations essentielles du projet



Une fois toutes les pages statiques construites à l'aide des maquettes faites au préalable sur Figma, il faut les rendre **dynamiques** et **interactives**. C'est là qu'intervient **javascript**. On commence par créer le système de **routage** qui permettra de **naviguer entre les pages**, puis les différents évènements comme le **chargement** d'une page ou l'interaction quand on **clique** sur un bouton par exemple.

Je n'oublie pas de **lier** le javascript et le css en les insérant dans le **<head>** de mon index.html (CSS) et en bas de mon **<body>** juste après mon **<footer>**.

```
</footer>
<script type="module" src="Router\router.js"></script>
<script src="js\script.js"></script>
</body>
```

J'inclus également dans mon **<head>** google font qui me permettra d'intégrer les différentes **polices** d'écriture.

```

<!DOCTYPE html>
<html lang="fr">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.googleapis.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Hind+Madurai:wght@300;400;500;600;700&family=Indie+Flower&family=Outfit:wght@100..900&display=swap" rel="stylesheet">
  <link rel="stylesheet" href="scss/main.css" />
  <title>Zoo Arcadia</title>
</head>

```

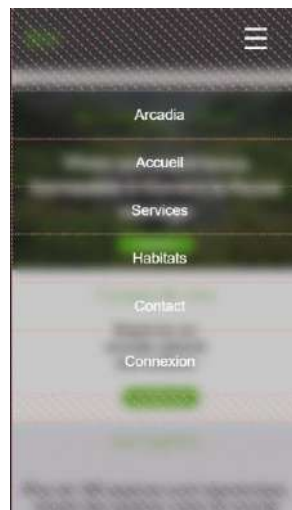
Pendant la création de mes pages je dois également penser au **responsive** de mon application afin qu'elle puisse s'adapter à **tous les formats**, dont le **mobile**. Pour cela, je me sers des **médias queries**, dans mon CSS.

```

/* ---MEDIA QUERIES PAGE HABITAT--- */
@media(max-width: 768px) {
  .imageArticleHabitat {
    display: none;
  }
}

```

Je vais donc adapter toutes mes pages ainsi que les parties importantes comme le menu de navigation que j'ai dû transformer en menu burger pour le format mobile.



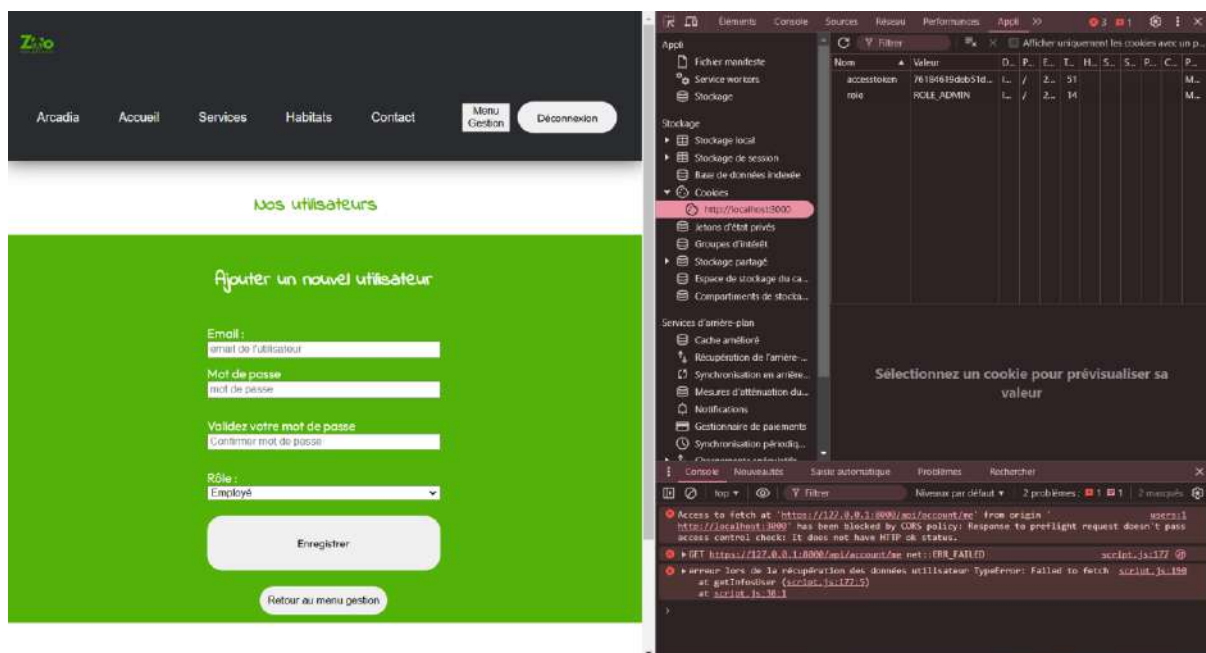
Le javascript et le css interviendront pour le rendre interactif et dynamique, en créant par exemple un évènement au clic qui fera simplement apparaître le menu de manière fluide.

```
//Mise en place du menu burger
const menuHamburger = document.querySelector(".menu-hamburger")
const navLinks = document.querySelector(".nav-links")

menuHamburger.addEventListener('click', () => {
  navLinks.classList.toggle('mobile-menu')
})
```

Il est important pour moi de garder un **code propre, simple, explicite** est bien présenté car j'ai pu avoir tendance à me perdre facilement au milieu de toutes ces lignes, j'ai dû apprendre à avoir certains **réflexes** afin d'être plus **efficace** et **productif**.

Grâce au serveur local je peux également regarder en temps réel les différentes modifications effectuées, et je n'hésite pas à regarder la **console dans le navigateur**, pour m'aider à identifier les problèmes grâce aux erreurs et le responsive.



Dans notre projet, nous savons qu'il y aura un **système de connexion et d'inscription** d'utilisateur pour le personnel du zoo, il faut donc créer une page permettant d'inscrire un utilisateur et une page permettant à celui-ci, de se connecter.



Il faudra prendre en compte **l'email**, **le mot de passe** et **le rôle** qu'il aura dans le système. Je crée donc un système de rôle, un système de connexion puis d'inscription en javascript.

Chaque rôle aura ses propres **droits d'accès**, **autorisations** à certaines pages et aux fonctionnalités.

Voici un exemple de fonction intégrée dans le front, qui permet de valider le formulaire d'inscription.

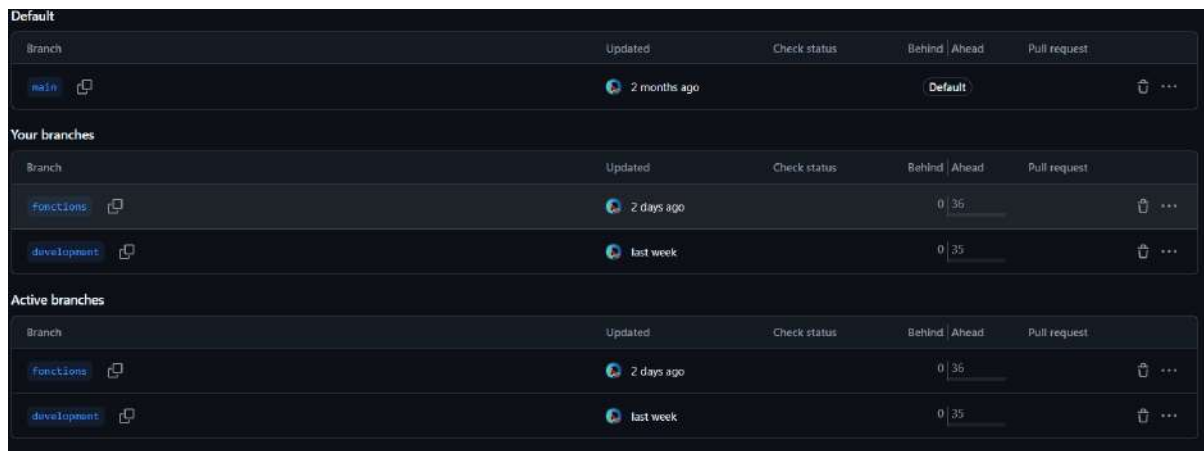
```

19 //Fonction de validation
20 function validateForm() {
21
22     const mailOk = validateMail(inputMail);
23     const passwordOk = validatePassword(inputPassword);
24     const passwordConfirmOk = validateConfirmationPassword(inputPassword, inputValidationPassword);
25
26     if (mailOk && passwordOk && passwordConfirmOk) {
27         btnValidation.disabled = false;
28         console.log("bouton activé");
29     } else {
30         btnValidation.disabled = true;
31         console.log("bouton désactivé");
32     }
33
34 }

```

Après avoir testé toutes les fonctionnalités et que tout est ok, je peux **merger** la branche "fonctions" sur la branche "development" et, avant déploiement, merger la branche "development" avec la branche "main". J'utiliserai **GIT/GITHUB** pour le **versionnage**, et le

déploiement continu. Dans un premier temps je fais un **commit** puis **push**, je sais que mon code est à jour, il me restera des fonctionnalités que j'ajouterai par la suite.



The screenshot shows a Git web interface with three sections: 'Default', 'Your branches', and 'Active branches'. Each section contains a table with columns: Branch, Updated, Check status, Behind | Ahead, and Pull request.

Branch	Updated	Check status	Behind Ahead	Pull request
main	2 months ago		Default	

Branch	Updated	Check status	Behind Ahead	Pull request
fonctions	2 days ago		0 36	
development	last week		0 35	

Branch	Updated	Check status	Behind Ahead	Pull request
fonctions	2 days ago		0 36	
development	last week		0 35	

Une fois le développement du frontend terminé, je peux éventuellement le mettre en ligne et faire des modifications au fur et à mesure grâce au déploiement continu. Mais je décide d'abord d'attaquer le backend, car je sais que je vais devoir ajouter des fonctionnalités dans mon frontend pour appeler l'api.

9. Le backend

Il est donc temps de passer au backend, côté serveur. Pour cette partie je me réfère au cours.

Avant de commencer, je configure la stack **AMP** : Apache2, MySql et Php, nécessaires pour héberger les projets php.

J'utiliserai le langage **php** et le framework **Symfony** qui lui-même intègre **Doctrine**, un **ORM** qui simplifie la gestion de base de données en mappant les objets Php aux tables SQL.

Je commence par observer l'**arborescence** et la structure des dossiers. Cela m'aide à prendre des repères mais aussi à comprendre quels fichiers sont importants.

Une fois que cela est fait, je lance le serveur Symfony pour tester si tout fonctionne correctement.

```
\Zoo_ArcadiaBackend> symfony server:start
```

Maintenant que tout fonctionne, il me faut créer la base de données. Pour cela je vais avoir besoin du diagramme de classe fait au préalable, installer et configurer Doctrine, puis configurer mon fichier **.env.local** afin de connecter ma base de données au projet.

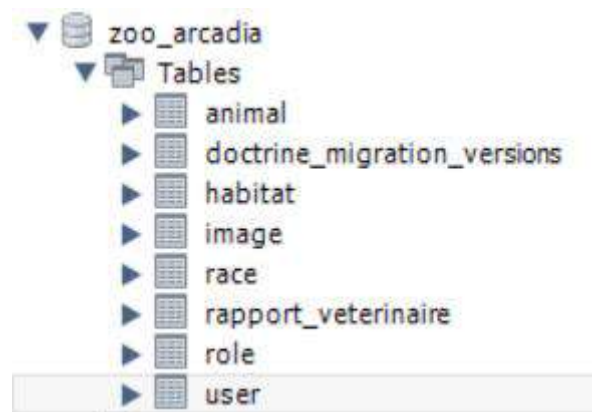
```
DATABASE_URL="mysql://arcadia:Arcadia123@127.0.0.1:3306/zoo_arcadia?serverVersion=15.1&charset=utf8mb4"
```

Pour cela, je modifie ma variable “**DATABASE_URL**” et mets toutes les informations nécessaires à la bonne configuration.

Je peux maintenant créer ma base de données. Pour cela, j'utilise la commande :

```
php bin/console make:entity user
```

Une fois créée, je fais la **migration** vers MySQL afin que toutes mes tables soient à jour.



Et je fais cela pour toutes les **entités**. Une fois toutes les tables ajoutées, il faudra y intégrer leur relation entre elles comme par exemple : les entités **Habitat** et **Animal**.

Pour cet exemple nous sommes sur une relation **Many To One**. Un habitat possède plusieurs animaux, mais un animal ne peut avoir qu'un seul habitat.

J'ai pu créer ma base de données, avec toutes les tables et leur relation. Je peux passer à la suite. Il faut maintenant créer une **interface utilisateur** où j'utiliserai la méthode **CRUD** qui signifie : **CREATE**, **READ**, **UPDATE** et **DELETE** qui me permettra de manipuler mes objets de manière simple et efficace.

Pour cela, je vais dans le dossier “**src**” puis dans “**Controller**” et je tape dans le terminal :

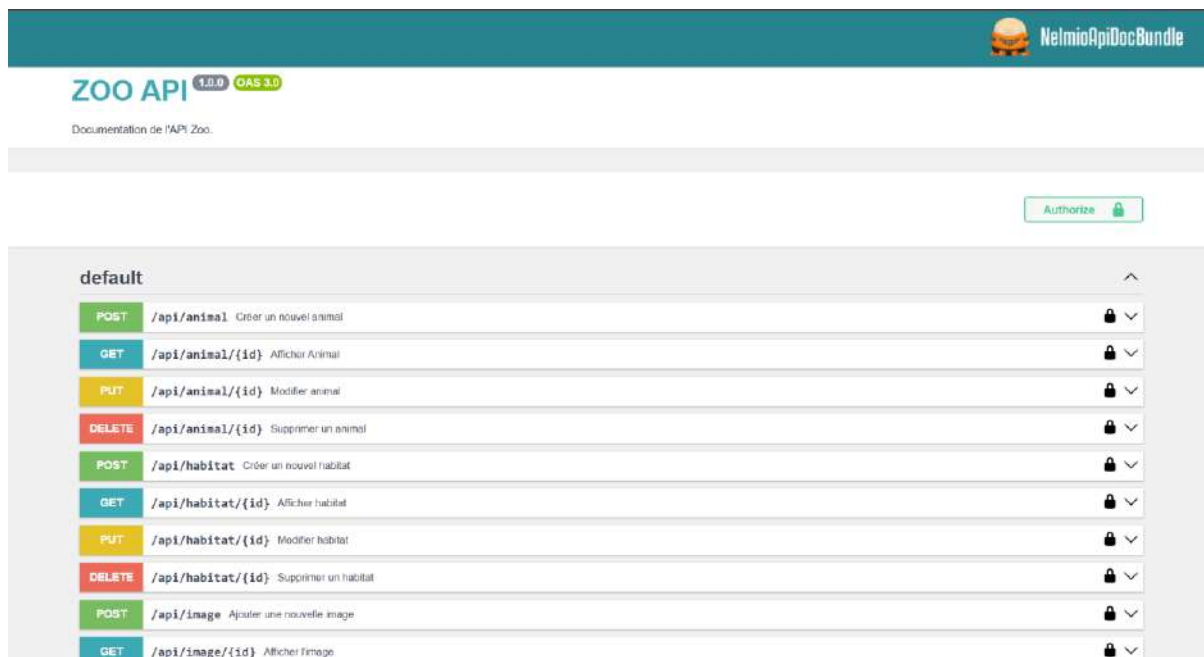
```
ArcadiaBackend> php bin/console make:controller security
```

Pour chaque controller, je créé des routes api et les configure et inclus les méthodes **POST**, **GET**, **PUT** et **DELETE** en fonction de mes besoins pour l'application. Grâce à Twig, le moteur de template html de symfony, je peux maintenant avoir un visuel de mon api.

Et y accéder en tapant dans le navigateur : <https://localhost:8000/api/doc>

On peut y voir tous les contrôleurs ainsi que les routes et les méthodes créés au préalable.

Nous avons aussi la possibilité de tester toutes nos requêtes directement depuis cette API.



Avant tout cela, je m'occupe en priorité de l'entité User qui est à part, puisqu'elle a les méthodes :

- getUserIdentifier
- getPassword
- getRoles
- getSalt
- eraseCredentials

Nous pourrons alors **s'inscrire** et **s'authentifier** de manière **sécurisée**. Il faudra alors **autoriser** ou **bloquer** l'accès aux routes en fonction du rôle de l'utilisateur. Par exemple : pour inscrire un nouvel utilisateur, seul l'administrateur du site y sera autorisé.

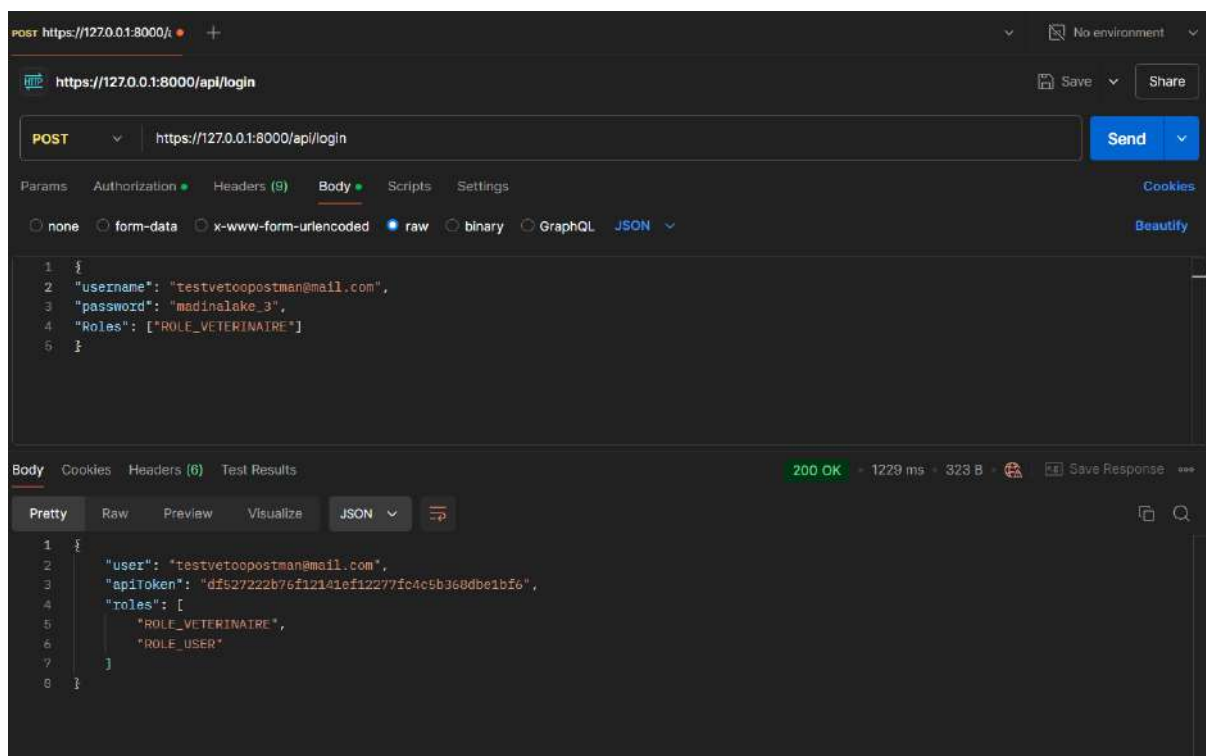

```

access_control:
- { path: ^/api/registration, roles: ROLE_ADMIN }
- { path: ^/api/login, roles: ROLE_USER }
- { path: ^/api/doc, roles: PUBLIC_ACCESS }
- { path: ^/api, roles: ROLE_USER }

```

Puis créer un **apiTokenAuthenticator**, qui sera récupérer à chaque requête par le pare-feu du gestionnaire de permission afin de **sécuriser l'api**.

Pour être sûr que mes requêtes et mes routes fonctionnent parfaitement, je décide d'utiliser **Postman**. Simple, intuitif, facile à utiliser il me permet de tester toutes mes requêtes sans passer par l'api, il me suffit simplement de lancer le serveur Symfony et d'écrire la route et la méthode souhaitée et de lancer la requête.



Sur cette image, je teste ma connexion avec un compte vétérinaire. J'inscris donc dans le body, les informations nécessaires à la connexion, je choisis la méthode POST puis la route. Postman me retourne bien une réponse en format **Json**: l'email, apiToken ainsi que son rôle respectif. Tout est OK.

Pour que l'api soit claire pour les développeurs qui souhaiteront l'utiliser, il faut la **documenter**. La documentation sert de guide sur les **fonctionnalités** offertes par l'api. Elle

indique ce que chaque **point d'entrée fait**, quelles informations sont **envoyées** et lesquelles sont **retournées**. Exemple avec le système de connexion ci-dessous.

```
#[OA\Post(
    path: "/api/login",
    summary: "Connecter un utilisateur",
    requestBody: new OA\RequestBody(
        required: true,
        description: "Données de l'utilisateur pour se connecter",
        content: new OA\JsonContent(
            type: "object",
            properties: [
                new OA\Property(property: "username", type: "string", example: "adresse@email.com"),
                new OA\Property(property: "password", type: "string", example: "Mot de passe")
            ]
        )
    ),
    responses: [
        new OA\Response(
            response: 200,
            description: "Connexion réussie",
            content: new OA\JsonContent(
                type: "object",
                properties: [
                    new OA\Property(property: "user", type: "string", example: "Nom d'utilisateur"),
                    new OA\Property(property: "apiToken", type: "string", example: "31a023e212f116124a36af14ea0c1c3806eb9378"),
                    new OA\Property(property: "roles", type: "array", items: new OA\Items(type: "string", example: "ROLE_USER"))
                ]
            )
        ]
    ]
)
```

Elle peut être sous forme **d'annotations** ou **d'attributs** suivant la version de php, pour ma part, j'ai dû convertir les annotations en attributs. Je procède donc de la même manière pour tous les contrôleurs, je documente mon CRUD.

Maintenant que j'ai testé mon api ainsi que toutes mes requêtes, que la documentation est faite et que tout est OK , d'une première part, je n'oublie pas de faire un commit puis push pour **mettre à jour mon repository distant** sur github puis, je vais tester mes différentes fonctionnalités.

Et pour cela je vais avoir besoin de **PHPUnit**, un frameworks de test **unitaire** et **fonctionnel** très puissant. J'installe le pack à l'aide de composer et la commande suivante :

```
aBackend> composer require --dev symfony/test-pack
```

Je fais dans un premier temps un test unitaire sur l'entité user afin de voir si, quand on crée un utilisateur, il reçoit bien une clé api.

```

0 references | 0 implementations
class UserTest extends TestCase
{

0 references | 0 overrides
public function testTheAutomaticApiTokenSettingWhenAnUserIsCreated(): void { //Récupère t-il l'api token?

$user = new User();
$this->assertNotNull(actual: $user->getApiToken());

}

```

J'initialise un nouvel objet User qui représente un nouvel utilisateur, puis utilise une méthode de PHPUnit : **assertNotNull**, qui vérifie que la méthode **getApiToken** nous renvoie une valeur qui n'est pas **null**, cela signifie qu'un token a bien été généré et assigné.

Pour ce type de test, il s'agit d'isoler une fonctionnalité, et de la tester indépendamment des autres. J'utilise la commande : **php bin/phpunit** pour lancer le test.

Je passe ensuite au test **fonctionnel**, qui servira plus à tester **un parcours utilisateur**.

Cette fois-ci, nous allons vérifier si la route **/api/doc**, qui est la page d'accueil de notre api, fonctionne bien.

```

10
11 public function testApiDocUrlIsSuccessful(): void
12 {
13     $client = self::createClient();
14     $client->followRedirects(followRedirects: false);
15     $client->request(method: 'GET', uri: '/api/doc');
16
17     self::assertResponseIsSuccessful();
18 }

```

La méthode **createClient** créer un client, nous désactivons la redirection puis, nous envoyons une requête HTTP de type GET à l'URL **/api/doc**. Pour finir l'assertion : **assertResponseIsSuccessful** vérifie que la réponse est bien réussie et qu'elle renvoie bien un code 200.

Maintenant que tous mes tests sont effectués. Il me faut tester la base de données. Pour cela je vais utiliser des **dataFixtures**. Je vais pouvoir créer des **données réalistes** afin de simuler des **manipulations de tables et des jeux de données**.

Dans un premier temps, j'installe le bundle de Doctrine à l'aide de la commande :

```

nd> composer require --dev orm-fixtures

```

Un dossier DataFixtures sera créé dans le dossier src. Je prends donc pour exemple l'entité **Animal** que je souhaite tester.

```
class AnimalFixtures extends Fixture
{
    0 references|0 overrides
    public function load(ObjectManager $manager): void //Création de datafixtures(FAUSSES DONNEES) avec la relation avec RapportVeterinaire
    {
        for ($i = 1; $i <= 20; $i++) {
            /** @var RapportVeterinaire $rapportVeterinaire */
            $animal = (new Animal())
                ->setPrenom(prenom: "Girafe $i")
                ->setEtat(etat: "Heureuse $i");

            $manager->persist(object: $animal);
            $this->addReference(name: "animal$i", object: $animal); //Créer la référence animal
        }

        $manager->flush();
    }
}
```

Ici la classe **AnimalFixtures** hérite de la classe **Fixture**, une classe abstraite du bundle de Doctrine. J'utilise la boucle **FOR** pour itérer vingt fois et créer 20 animaux différents en base de données. À chaque tour de boucle, un nouvel animal est créé. Je crée une nouvelle instance "**Animal**" avec les méthodes **setPrenom**, qui attribuera un prénom unique au format "Girafe 1", "Girafe 2", puis la méthode **setEtat**, qui attribuera un état également unique "Heureuse 1", "Heureuse 2" etc .

Je prépare l'objet animal à être envoyer en base de données avec "**persist**".

Je stock une référence à l'objet **\$animal**, cela est utile quand d'autres fixtures en dépendent, comme une relation entre "**RapportVeterinaire**" et "**Animal**".

Je pousse ensuite toutes les entités persistées en base de données.

Je sais que ma fixture "**AnimalFixtures**" est en relation avec ma fixture "**RapportVeterinaireFixtures**". Il faut donc que je récupère la référence "**AnimalFixtures**" dans "**RapportVeterinaireFixtures**". Comme ci-dessous :

```

5 references | 0 implementations
class RapportVeterinaireFixtures extends Fixture implements DependentFixtureInterface
{
    0 references | 0 overrides
    public function load(ObjectManager $manager): void //Création de datafixtures(FAUSSES DONNEES)
    {
        for ($i = 1; $i <= 20; $i++) {
            /** @var Animal $animal */
            $animal = $this->getReference(name: "animal" . random_int(min: 1, max: 20)); //Récupère la référence animal

            $rapportVeterinaire = (new RapportVeterinaire())
                ->setDate(date: date_create_immutable())
                ->setDetail(detail: "a mangé 3 repas $i")
                ->setAnimal(animal: $animal);

            $manager->persist(object: $rapportVeterinaire);
        }

        $manager->flush();
    }
    0 references | 0 overrides
    public function getDependencies(): array
    {
        return [AnimalFixtures::class];
    }
}

```

Nous voyons bien dans cet exemple, que l'on récupère la référence **"animal"** avec la méthode **getReference**. Les deux entités sont bien associées, chaque animal créé aura bien un rapport vétérinaire.

J'ai pu donc tester ma base de données et les différentes relations entre elles en simulant des données réalistes.

10. Lier le Frontend et le Backend

Maintenant que je suis bien avancé dans mon développement, il faut que le front puisse appeler le backend via l'api et interagissent ensemble.

Je vais commencer par tester une inscription depuis mon formulaire d'inscription de la page gestionnaire d'utilisateur. Pour être sûr, je teste de nouveau mes requêtes sur l'api mais aussi sur Postman. Tout est OK.

Dans mon Frontend, j'utiliserai la fonction **FETCH** pour faire mes requêtes HTTP **asynchrones**.

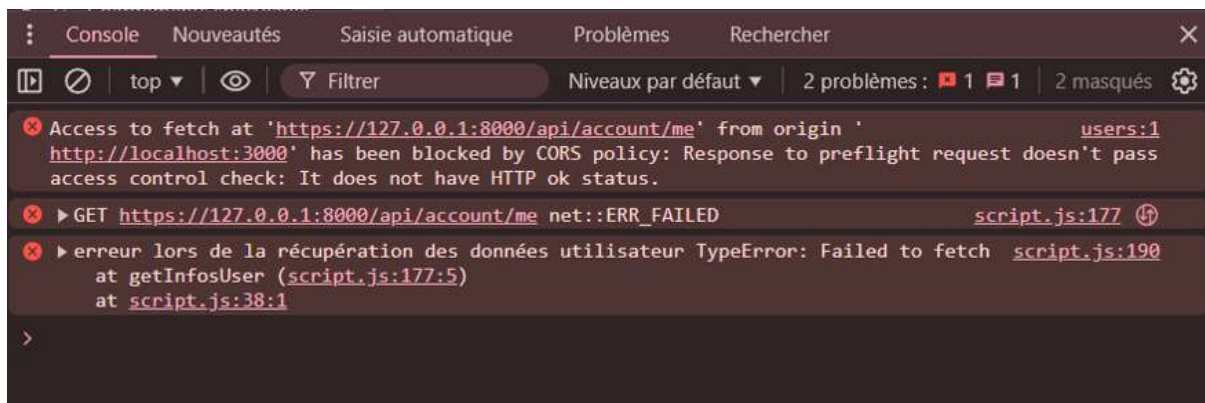
```

164
165 fetch("https://127.0.0.1:8000/api/registration", requestOptions)
166 .then(response => {
167     if(response.ok){
168         return response.json();
169     }
170     else{
171         alert("Erreur lors de l'inscription");
172     }
173 })
174
175 .then(result => {
176     alert("Bravo vous êtes maintenant inscrit, vous pouvez vous connecter");
177     document.location.href="/signin";
178 })
179 .catch(error => console.log('error', error));
180 }

```

Étant en local, j'indique dans mon fetch, l'url de mon api avec la route vers l'inscription utilisateur. Si tout est OK il me retourne une réponse en **Json** et redirige l'utilisateur sur la page de connexion sinon, il me retourne un message d'erreur.

Durant cette partie du développement j'ai pu être confronté à un problème récurrent. L'erreur **CORS**. En effet, étant donné que les navigateurs imposent ce système de sécurité pour protéger les utilisateurs et de restreindre les requêtes effectuées venant de différentes origines, je me suis retrouvé bloqué. Mon front et mon Back sont tous deux, déjà, sur deux protocoles différents, mais aussi deux adresses différentes.



Pour pallier à cette erreur, plusieurs solutions s'offrent à moi :

- Vérifier la configuration de mon fichier **nelmio_cors.yaml** dans mon backend :

```
! nelmio_cors.yaml x $ .env.local
config > packages > ! nelmio_cors.yaml
1  nelmio_cors:
2    defaults:
3      origin_regex: true
4      allow_credentials: true
5      allow_origin: ['%env(CORS_ALLOW_ORIGIN)%']
6      allow_methods: ['GET', 'OPTIONS', 'POST', 'PUT', 'PATCH', 'DELETE']
7      allow_headers: ['Content-Type', 'Authorization', 'X-Requested-With']
8      expose_headers: ['Link']
9      max_age: 3600
10   paths:
11     '/': null
12
```

- Vérifier mon fichier **.env.local** dans mon backend :

```
###> nelmio/cors-bundle ###
CORS_ALLOW_ORIGIN='^https?://(localhost|127\.0\.0\.1)(:[0-9]+)?$'
```

D'après le paramétrage de ses deux fichiers, l'origine bien définie grâce à son **REGEX**, le navigateur devrait pouvoir accepter mes requêtes depuis le frontend.

- Vérifier la syntaxe de l'url api que j'ai inscrite dans mon frontend :

```
fetch("https://127.0.0.1:8000/api/registration", requestOptions)
```

J'ai finalement opté pour une solution un peu plus radicale, qui est de désactiver ce système directement dans le navigateur grâce à une manipulation dans l'invite de commande.

Chose faite, je peux tester mon inscription, mais aussi ma connexion.

Les informations utilisateur sont bien récupérées, stockées en cookie et envoyées en base de données.

Pour la connexion, on récupère bien le **token** et le **rôle** de l'utilisateur grâce à la fonction **getInfosUser** dans mon **script.js**.


```

//Fonction pour récupérer les informations utilisateur
function getInfosUser(){
  console.log("récupération des infos de l'utilisateur");
  let myHeaders = new Headers();
  myHeaders.append("X-AUTH-TOKEN", getToken());

  let requestOptions = {
    method: 'GET',
    headers: myHeaders,
    redirect: 'follow'
  };

  fetch("https://127.0.0.1:8000/api/account/me", requestOptions)
    .then(response =>{
      if(response.ok){
        return response.json();
      }
      else{
        console.log("Impossible de récupérer les informations utilisateur");
      }
    })
    .then(result => {
      if (result) {
        console.log("Informations utilisateur :", result); //Renvoie les infos utilisateurs dans la console
      }
    })
    .catch(error =>{
      console.error("erreur lors de la récupération des données utilisateur", error);
    });
}

```

Mes requêtes sont donc authentifiées et sécurisées grâce aux jetons.

11. Le déploiement

Arrivant bientôt en fin de projet, mes repos distants étant à jour, je n'hésite pas à retester mon api pour vérifier si tout est OK.

Il faut maintenant passer à la mise en ligne de l'application.

Pour la partie front, j'utiliserai l'hébergeur **Always Data** que j'ai pu tester durant les cours. Il est flexible, polyvalent et est compatible avec un grand nombre de base de données. De plus, il propose un hébergement gratuitement de 100 Mo.

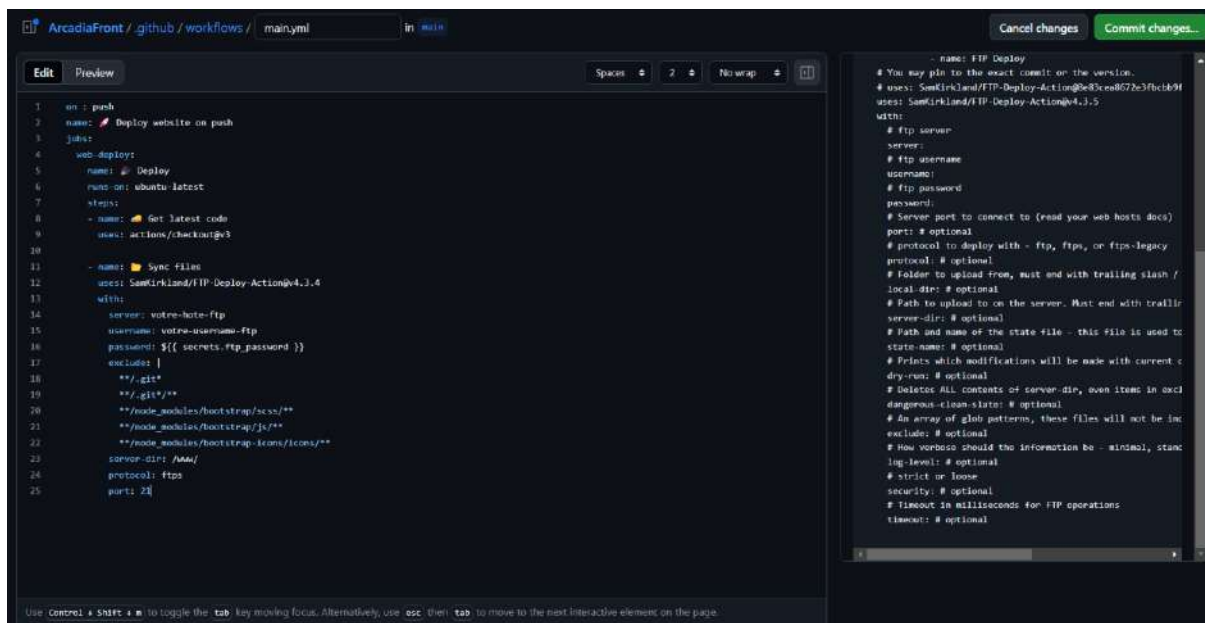
Je commence donc par créer un compte avec une adresse email et un mot de passe. Une fois crée je donne un nom à mon site puis le configure.

Je me connecte avec l'hôte FTP, le nom du site puis son mot de passe, et fait glisser tous mes fichiers dans le dépôt distant. Une fois que c'est fait, je peux maintenant accéder mon site en ligne en cliquant sur son lien.

parcarcadia.alwaysdata.net

Maintenant que mon site est disponible, je souhaite pouvoir faire des mises à jour en continu tout en gardant mon site en ligne.

Pour assurer le déploiement continu, j'utiliserai **Github** à chaque mise à jour du code qui sera déployée directement en production.



```
on: push
name: Deploy website on push
jobs:
  web-deploy:
    name: Deploy
    runs-on: ubuntu-latest
    steps:
      - name: Get latest code
        uses: actions/checkout@v3

      - name: Sync files
        uses: SamKirkland/FTP-Deploy-Action@v4.3.4
        with:
          server: votre-hote-ftp
          username: votre-username-ftp
          password: ${{ secrets.ftp_password }}
          exclude: |
            **.git*
            **/node_modules/bootstrap/scss/**
            **/node_modules/bootstrap/js/**
            **/node_modules/bootstrap-icons/**
          server-dir: /www/
          protocol: ftps
          port: 21
```

Je crée un **workflow** dans l'onglet **ACTION**. Un fichier **main.yaml** se crée. J'intègre mon code de configuration puis je fais un "**Commit Changes**". Cela rajoutera le fichier github dans mon vs code. À chaque fois que je modifierai mon code et à chaque push vers mon dépôt distant, la mise à jour du déploiement se fera automatiquement et mon site aura les modifications apportées.



Le déploiement du backend, quant à lui se fera sur **Platform.sh**, une plateforme d'hébergement cloud de déploiement continu qui me permettra de gérer facilement mon environnement de production.



Pour commencer, je crée un compte sur la plateforme.

Puis pour préparer le projet, j'entre dans mon terminal Vs Code : **Symfony project:init**

De nouveaux fichiers s'ajoutent au projet. Dans le dossier "**platform/**" je vais configurer le fichier "**services.yaml**".

```
$.env.local  AnimalController.php  ! services.yaml X
.platform > ! services.yaml
1  database:
2      type: mysql:10.6
3      disk: 512
4
```

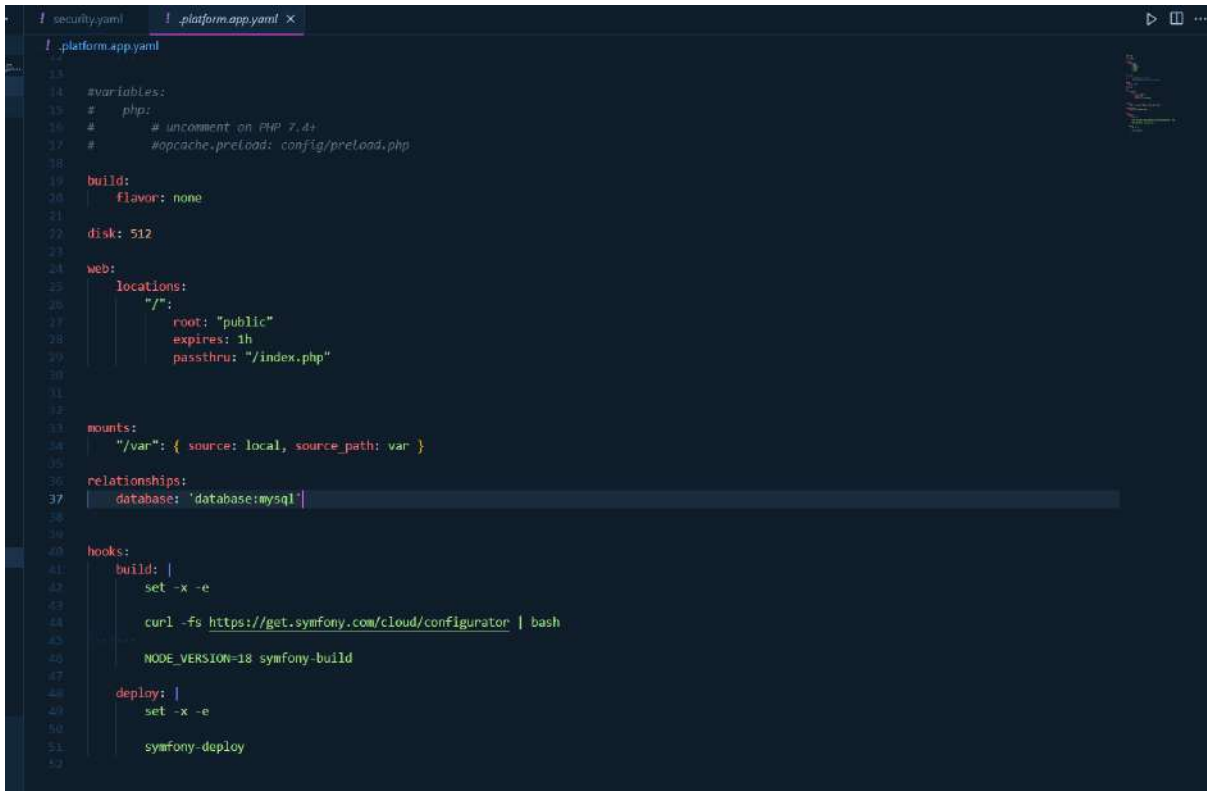
Je pense à vérifier que la **version** de ma base de données est la même partout, notamment dans le fichier "**doctrine.yaml**".

```
config > packages > ! doctrine.yaml
1  doctrine:
2      dbal:
3          url: '%env(resolve:DATABASE_URL)%'
4          server_version: '10.6'
5          # IMPORTANT: You MUST configure your server version,
6          # either here or in the DATABASE_URL env var (see .env file)
7          #server_version: '16'
```

Mais également mon fichier "**.env**".

```
DATABASE_URL="mysql://arcadia:Arcadia123@127.0.0.1:3306/zoo_arcadia"
```

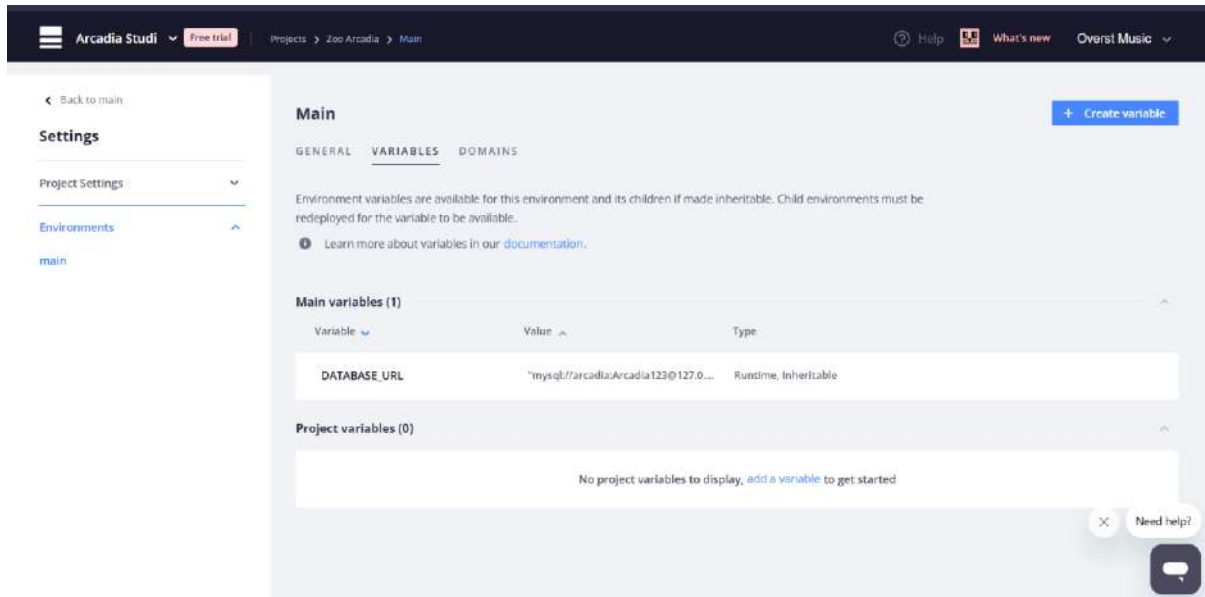
Je vais ensuite sur le fichier de configuration du déploiement “**platform.app.yaml**” afin d’y introduire la relation avec la base de données.



```
13
14 #variables:
15 #   php:
16 #       # uncomment on PHP 7.4+
17 #       #opcache.preload: config/preload.php
18
19 build:
20   flavor: none
21
22 disk: 512
23
24 web:
25   locations:
26     "/":
27       root: "public"
28       expires: 1h
29       pass thru: "/index.php"
30
31
32
33 mounts:
34   "/var": { source: local, source_path: var }
35
36 relationships:
37   database: 'database:mysql'
38
39
40 hooks:
41   build: |
42     set -x -e
43
44     curl -fs https://get.symfony.com/cloud/configurator | bash
45
46     NODE_VERSION=18 symfony-build
47
48   deploy: |
49     set -x -e
50
51     symfony-deploy
52
```

Je vais maintenant retourner sur platformSh et créer **ma variable d’environnement** : **DATABASE_URL**, dans les paramètres de “**MAIN**”. Ma base de données passera en priorité par cette variable pour la connexion.

Cette configuration permet d’éviter des erreur 500 ou 401 lorsque que je teste mon api une fois déployée.



Une fois chose faite, je vais lier mon projet à la plateforme. Pour cela je vais entrer la commande : **symfony cloud:project:create --title Arcadia Studi --set-remote** , afin de créer le projet.

Puis je vais le **pusher sur le serveur distant** avec la commande : **symfony cloud:environment:push** .

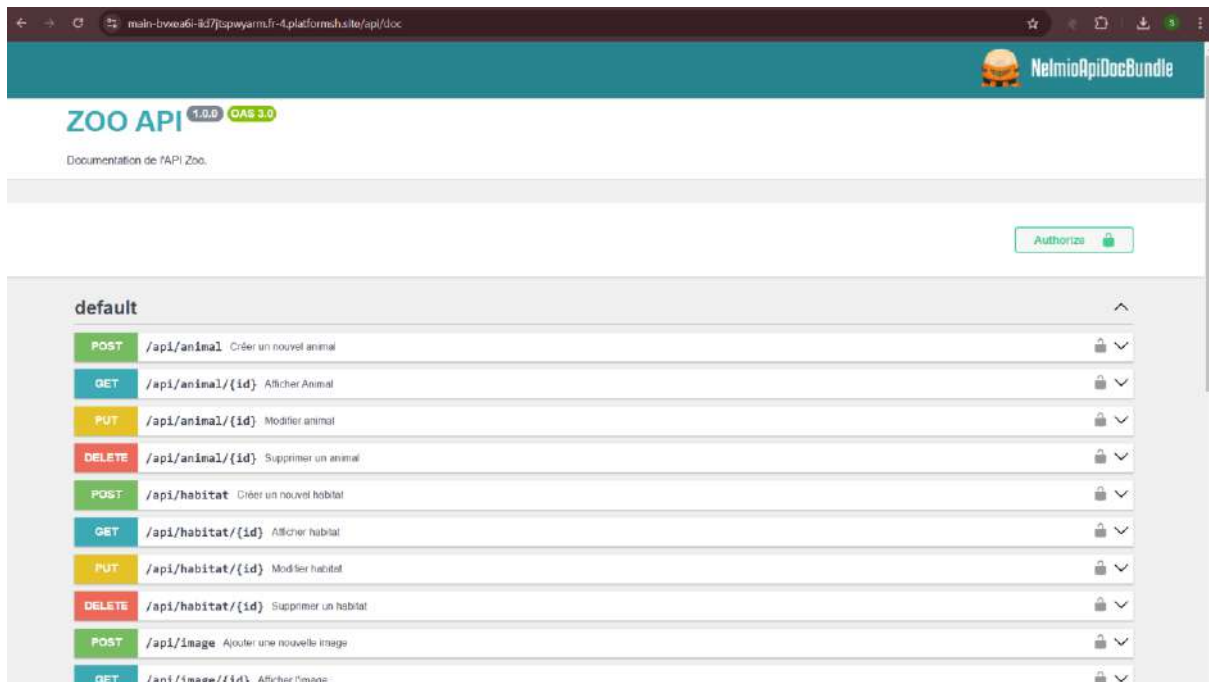
Je peux maintenant, à chaque mise à jour de mon code, utiliser cette commande pour le déploiement continu.

Le lien de mon backend est maintenant disponible, quand je clique dessus il m'ouvre une page :



Bienvenue sur votre accueil !

Je peux aussi accéder à mon api en rajoutant à l'url : **/api/doc**, puis la tester.



12. Fin du projet

En conclusion de ce projet. Je dirais qu'il m'a beaucoup apporté en termes de connaissances et de compétences. Il a d'avantage éveillé ma curiosité et ma passion pour le codage.

Il m'a appris à être patient face à un problème et d'avoir une prise de recul nécessaire à sa résolution.

C'est un domaine riche et varié qui évolue de manière constante. Il y a et il y aura, selon moi, toujours à apprendre. D'où l'importance de faire des veilles technologiques et de rester à l'affût des dernières tendances et mises à jour majeures. Il me reste encore beaucoup à apprendre et à améliorer sur mon site, je reste donc actif sur mon processus d'apprentissage.