



Dossier de Projet

Survival-Parc : Parc d'attraction

Projet réalisé dans le cadre
de la présentation au Titre
Professionnel Développeur
Web & Web Mobile

Présenté et soutenu par Aline COATANOAN



SOMMAIRE

INTRODUCTION -----	p4
PRÉSENTATION DU PROJET -----	p5
CAHIER DES CHARGES -----	p6
A. Conceptualisation de l'application -----	p6
Minimum Viable Product -----	p6
Workflow de l'application -----	p8
B. Utilisateur et User Story -----	p8
Public visé et rôles -----	p8
User Stories -----	p9
C. Mise en place de l'application -----	p10
Routes -----	p10
MCD (Modèle Conceptuel de Données) -----	p11
MLD (Modèle Logique) -----	p11
MPD (Modèle Physique) -----	p12
Dictionnaire des données -----	p12
Wireframes -----	p13
Charte graphique -----	p14
SPECIFICATIONS TECHNIQUES -----	p15
A. Technologies Front-end -----	p15
B. Technologies Back-end -----	p16
C. Sécurité de l'application -----	p18
ORGANISATION -----	p18
A. Explication du choix d'un projet en autonomie -----	p18
B. Méthodologie de travail -----	p19



COMPETENCES DU REFERENTIELS -----	p21
A. Front-end -----	p21
CP 1 : Installer et configurer son environnement de travail -----	p21
CP 2 : Maquetter des interfaces utilisateur -----	p23
CP 3 : Réaliser des interfaces utilisateur statiques -----	p25
CP 4 : Développer la partie dynamique des interfaces utilisateur web ou web mobile -----	p26
B. Back -----	p28
CP 5 : Mettre en place une base de données relationnelle -----	p28
CP 6 : Développer des composants d'accès aux données SQL -----	p30
CP 7 : Développer des composants métier côté serveur -----	p33
CP 8 : Documenter le déploiement d'une application dynamique -----	p37
ELEMENTS DE SECURITE -----	p38
A. Authentification sécurisée (JWT TOKEN) -----	p38
B. Contrôle de champs (Joi) -----	p40
C. Gérer les sessions et renforcer la sécurité contre XSS et CSRF (Cookies) -----	p41
D. Hachage du mot de passe -----	p43
JEU D'ESSAI -----	p44
A. Test unitaire -----	p44
B. Mise à jour d'un profil -----	p45
VEILLE SÉCURITÉ -----	p46
A. Veille technologique -----	p46
B. Résolution de problème -----	p46
CONCLUSION -----	p49
ANNEXES -----	p50

1. INTRODUCTION

Après dix années enrichissantes en tant qu'éducatrice spécialisée, j'ai ressenti le besoin d'un renouveau professionnel. Curieuse des possibilités qu'offre le monde numérique, j'ai entamé une réflexion sur ma reconversion. Grâce à l'accompagnement d'un Conseiller en Évolution Professionnelle, j'ai découvert l'univers du développement web.

Pour me plonger dans cet univers, j'ai commencé par explorer les bases avec le HTML et le CSS. Cette première approche a confirmé mon intérêt et mon envie de me former sérieusement à ce métier. Après avoir recherché une formation adaptée à mes besoins et à ma situation, j'ai choisi O'Clock. Avec le soutien de mon employeur et le financement obtenu via Transition Pro, j'ai pu intégrer cette formation et me lancer pleinement dans cette aventure.

Suite à la formation, j'ai travaillé sur un projet concret : la création d'une application web pour un parc à thème fictif, appelée Survival-Parc. Ce projet a été l'occasion de mettre en pratique les connaissances acquises tout en répondant à des problématiques réelles du domaine du développement.

Le projet Survival-Parc vise à développer une application web permettant aux utilisateurs de réserver des activités au sein d'un parc à thème. Les principales fonctionnalités du projet incluent :

- L'affichage détaillé des activités disponibles dans le parc, avec des options de recherche et de filtre (par catégorie).
- Un système de réservation permettant aux utilisateurs de réserver en ligne un nombre de billets d'entrée et des nuits d'hôtel pour le parc à une date donnée.
- La mise en place d'un système de connexion, inscription, et gestion du profil utilisateur.
- L'affichage détaillé des réservations effectuées par l'utilisateur avec la possibilité d'annuler une réservation jusqu'à 10 jours avant la date d'entrée.
- Un back-office avec la liste des réservations, des activités, des hôtels, permettant ainsi à l'administrateur de gérer facilement les différentes facettes du parc.

Ce projet m'a permis de travailler sur l'intégration de ces fonctionnalités, tout en garantissant une expérience utilisateur fluide et une gestion sécurisée des données sensibles.



2. PRÉSENTATION DU PROJET

- Introduction

Survival Parc vous fait plonger dans un univers captivant et immersif où l'aventure et l'adrénaline règnent en maître. Dans ce parc unique en son genre, le but est d'être transporté dans un monde post-apocalyptique ravagé par un mystérieux virus zombie. Inspiré de l'ambiance des grandes sagas telles que *The Last of Us*, le parc recrée une zone de quarantaine où chaque détail vous immerge dans cet univers troublant mais exaltant.

Survival Parc offre une variété d'attractions pour satisfaire les amateurs de sensations fortes et les fans de récits post-apocalyptiques.

Le parc, interdit aux moins de 16 ans pour préserver l'intensité de l'expérience, propose des billets à la journée, mais aussi un hébergement dans l'un de ses deux hôtels à l'ambiance post-apocalyptique.

- Objectif

L'objectif principal est de fournir aux visiteurs un aperçu captivant du parc, tout en leur offrant la possibilité de créer un compte, un profil et de réserver leurs billets et leur nuit d'hôtel en ligne de manière pratique et sécurisée, fournir des descriptions des attractions, des hôtels, le plan du parc, les horaires, l'accès au parc.

Le gérant doit pouvoir être autonome, il y a donc une partie back-office pour lui laisser la main sur le site, lui permettant de voir toutes les réservations, et modifier les informations concernant les animations et les hôtels.



3. CAHIER DES CHARGES

A. Conceptualisation de l'application

Minimum Viable Product (MVP)

Les MVP regroupent tous les éléments que le site peut offrir, aussi bien pour les utilisateurs que pour les administrateurs.

- Fonctionnalités utilisateur :

→ Page d'accueil :

- ◆ Accès au menu, aux boutons inscription/connexion.
- ◆ Accès vers l'ensemble des animations du parc
- ◆ Accès vers les hôtels
- ◆ Accès aux informations utiles (contact, adresse)
- ◆ Accès à une barre de recherche

→ Page de Réservation de billets :

- ◆ Accès aux tickets journées avec choix de la date et du nombre de personnes.

→ Pages de Présentation des attractions :

- ◆ Des pages dédiées décrivant chaque attraction avec des photos, et détails pratiques (durée, intensité, règles).

→ Pages des hôtels :

- ◆ Une section pour découvrir les hôtels (photos, ambiances, types de chambres) et réserver un séjour avec lien vers la réservation.

→ Page d'informations pratiques :

- ◆ Accès au plan du parc avec le détail des emplacements des animations.
- ◆ Une FAQ détaillant les règles d'âge, les consignes de sécurité, et les services proposés (restaurants, parkings, etc.).
- ◆ Horaires du parc
- ◆ Accès au parc pour indiquer la route, les parkings.

→ Page profil :

- ◆ Accès aux informations de son profil.
- ◆ Accès à l'historique de ses réservations

→ Page panier :

- ◆ Qui pourra permettre un paiement en ligne (piste d'amélioration)

- **Fonctionnalités administrateur :**

→ Gestion des contenus :

- ◆ Interface permettant d'ajouter/modifier les attractions, événements ou promotions (piste d'amélioration).
- ◆ Gestion des hôtels et de leurs informations.

→ Analyse des données :

- ◆ Vu sur les réservations (piste d'amélioration : statistiques)

- **Les évolutions possibles :**

→ Analyse des données :

- ◆ système de messagerie asynchrone entre les utilisateurs et les administrateurs.

→ Notation :

- ◆ possibilité de noter une activité, affichage pour chaque activité de la notation moyenne et du nombre de notes.

→ Commentaires :

- ◆ possibilité de commenter une activité (avec modération en back-office).

→ Paiement :

- ◆ Système de paiement sécurisé en ligne.



Workflow de l'application

Un workflow montre comment les utilisateurs et administrateurs interagissent avec le site. J'ai effectué ce schéma avec Canva. J'ai mis un code couleur afin de différencier ce qui est accessible en tant que visiteur, utilisateur connecté et administrateur. (Cf ANNEXE 1)

B. Utilisateur et User Story

Public visé et rôles

Survival Parc cible un public spécifique qui partage un intérêt pour les expériences immersives, les sensations fortes, et les scénarios fictifs. Voici le genre de public visé pour cette expérience.:

- Les passionnés de thématiques post-apocalyptiques et de zombies :

Personnes recherchant une immersion totale dans un univers fictif. Fans de séries, jeux vidéo, et films liés aux zombies (*The Last of Us*, *World War Z*, etc.).

- Les amateurs de sensations fortes et d'activités immersives :

Groupes d'amis ou familles avec adolescents de plus de 16 ans, chercheurs d'adrénaline et d'expériences hors du commun..

- Les groupes d'amis et sorties de groupes :

Groupes célébrant des événements comme des anniversaires ou des clubs et associations recherchant des activités originales.

Le rôle des utilisateurs :

- Visiteur / utilisateur non connecté : Il peut accéder aux informations liées au parc (attractions, hôtels, tarifs, contact, informations pratiques, informations utiles).
- Utilisateur connecté : Il a accès aux mêmes informations et peut également accéder à son compte avec ses informations personnelles (nom, prénom, coordonnées), et



l'historique de ses réservations. Il peut également faire des réservations s'il a créé un profil. Il peut annuler sa réservation jusqu'à 10 jours avant la date.

User Stories

Les user stories sont des descriptions simples et concises des besoins ou attentes d'un utilisateur vis-à-vis d'une fonctionnalité. Ici, j'ai réalisé un user stories décrivant les attentes d'un visiteur non inscrit, d'un utilisateur inscrit et d'un administrateur.

Ex :

En tant que	Je veux	Afin de
Visiteur non inscrit	parcourir les animations	voir les informations sur l'animation
Visiteur non inscrit	créer un compte	réserver / accéder à mon profil
Visiteur non inscrit	pouvoir effectuer une recherche	trouver des informations précises
Visiteur non inscrit	pouvoir contacter le support client	poser des questions
Visiteur non inscrit	accéder aux informations pratiques	connaître les horaires et l'accès au parc

(Cf : ANNEXE 2)

D. Mise en place de l'application

Routes

Les routes API définissent des endpoints accessibles via des méthodes HTTP pour effectuer des opérations CRUD. Ces routes permettent de communiquer entre le front et le back : Par exemple, lorsqu'un utilisateur clique sur "Réserver", le front envoie une requête au back via un endpoint comme /reservation. L'API traite la requête et met à jour la base de données.

Ici nous avons :

- l'URL : Point d'accès à la route
- Les méthodes HTTP : Action effectuée lors d'une requête. Les principales méthodes sont GET, POST, PUT, DELETE.
- Les controllers : Il contient la logique associée à une fonctionnalité. Il gère l'aspect dynamique de l'application. A partir de la requête, il récupère les données dans les Models, les injecte dans la vue (réponse JSON), et envoie la réponse produite. C'est dans les controllers qu'on retrouve les opérations CRUD.
- Les méthodes : Fonctions qui permettent de manipuler des données ou d'interagir avec une application. Elles servent à exécuter des tâches spécifiques dans une base de données.
- Les commentaires : Définit la requête.

URL	Méthode HTTP	Controller	Méthode	Commentaire
api/auth/login	post	authController	findOne	se connecter
api/auth/logout	post	authController	findByPk	se déconnecter
api/user/register	post	userController	create	créer un compte
api/user	get	userController	findAll (where: { role: "admin" },)	récupérer les users/admins
api/user/:id	get	userController	findByPk	récupérer un user

(Cf Annexe 3)

MCD (Modèle Conceptuel de Données)

Le MCD est une représentation abstraite de la base de données qui décrit les entités et les relations entre elles de manière conceptuelle, sans se préoccuper de la façon dont elles sont stockées physiquement. Dans ce MCD nous trouvons 5 entités (utilisateur, profil, hotel, réservations, animations) avec chacune leurs attributs (nom, prénom, description, prix...). Les cardinalités (peut, défini, possède) définissent les relations entre les entités et spécifient le nombre d'instances (0,1,n) d'une entité qui peuvent être associées à une instance d'une autre entité.

1-1 : Une instance d'une entité A peut être associée à une et une seule instance de l'entité B, et vice-versa.

1-N : Une instance d'une entité A peut être associée à plusieurs instances de l'entité B, mais une instance de l'entité B ne peut être associée qu'à une seule instance de l'entité A.

N-N : Plusieurs instances de l'entité A peuvent être associées à plusieurs instances de l'entité B, et inversement. Cela nécessite une table de liaison pour représenter cette relation.

0 : Une instance de l'une des entités peut exister sans être associée à une instance de l'autre entité.

(Cf Annexe 4)

MLD (Modèle Logique)

Le MLD est une représentation plus détaillée qui se rapproche du modèle physique de la base de données. C'est une vue logique où l'on définit les tables qui correspondent aux entités, et chaque table contient des colonnes avec des champs qui correspondent aux attributs des entités, nous notons les ID et les clés étrangères. Ici nous trouvons la table de liaison : ProfileHotel (relation many-to-many), avec les champs, profileId (clé étrangère vers Profile), hotelId (clé étrangère vers Hotel). (Cf Annexe 5)

La table de liaison permet de gérer une relation entre deux tables. Là où la table hôtel contient des informations fixes uniquement modifiable par l'administrateur, la table de liaison permet à un profil de réserver l'hôtel avec des informations modulables (prix, dates, nombre de personnes).

MPD (Modèle Physique des Données)

Le MPD se concentre sur la représentation physique des données dans un système de gestion de base de données (SGBD) tel que MySQL. Contrairement au MCD ou au MLD, qui décrivent les entités et leurs relations d'une manière abstraite, le MPD spécifie comment ces entités et relations seront stockées dans la base de données. Par exemple on retrouve AUTO-INCREMENT qui permet d'augmenter et générer automatiquement les id, VARCHAR qui est une chaîne de caractère, NOT NULL qui signifie que la colonne ne peut pas être vide...

ex de MPD pour la table User :

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    firstName VARCHAR(30) NOT NULL,
    lastName VARCHAR(30) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    role ENUM('admin', 'user') NOT NULL DEFAULT 'user',
    createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

Dictionnaire des données

Le dictionnaire de données est comme un guide ou une fiche descriptive de tout ce qui se trouve dans la base de données. Il explique quelles données sont stockées, où elles se trouvent (dans quelles tables), et comment elles sont utilisées (par exemple, leurs types, leurs relations).

C'est une sorte de mode d'emploi pour mieux comprendre la base de données et éviter les erreurs. On le crée à partir du MLD.



Le dictionnaire de données contient des informations détaillées sur chaque élément de donnée dans une base de données, telles que :

- Le nom des tables : Les entités ou les objets de données dans le système.
- Le nom des colonnes : Les attributs ou propriétés de chaque entité.
- Le type de données des colonnes : Par exemple, VARCHAR, INT, DATE, etc.
- Les contraintes : Les règles associées aux données, telles que les clés primaires, les clés étrangères, les contraintes d'unicité, les valeurs par défaut, et les validations (par exemple, validation de longueur, validation de format, etc.).
- Les descriptions des données : Les explications sur la signification ou l'usage des colonnes, en particulier pour des colonnes complexes ou spécifiques.

(Cf Annexe 6)

Wireframes

Un wireframe est une représentation visuelle simplifiée de l'interface utilisateur d'une application. Il sert de plan ou de croquis pour montrer l'agencement des éléments sur les pages, comme les menus, les boutons, les images, et les textes.

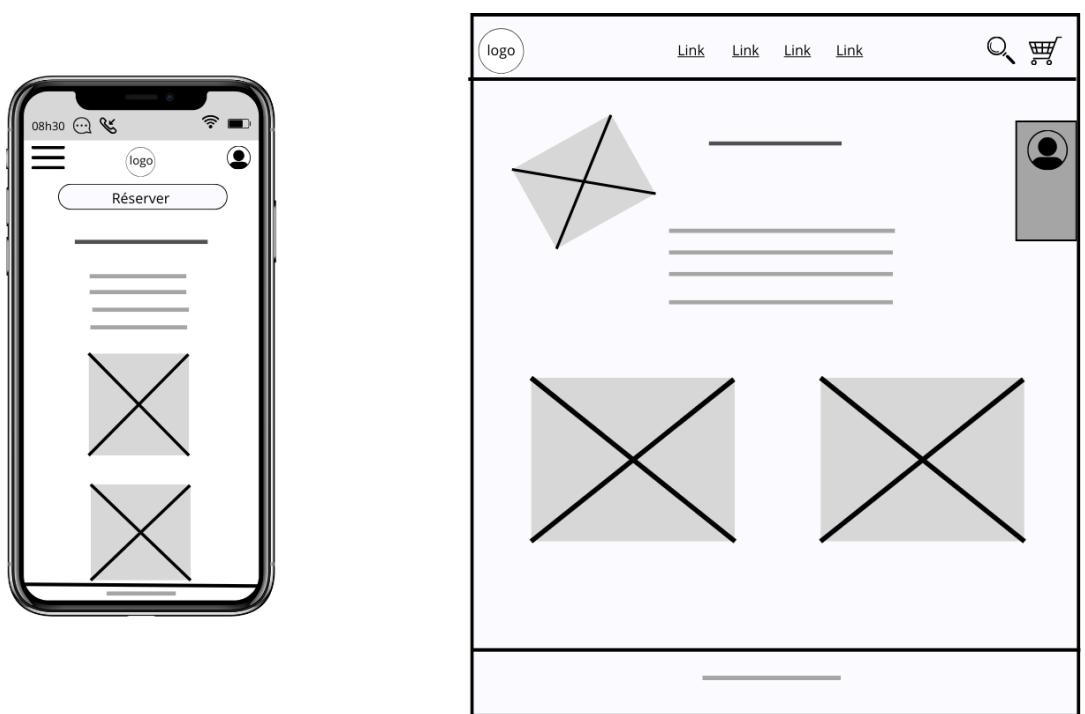
Pour créer un wireframe de survival parc, j'ai décidé des informations et fonctionnalités essentielles pour chaque page. J'ai déterminé comment organiser les éléments pour guider efficacement l'utilisateur et pensé aux parcours utilisateurs, aux liens entre les pages, et à la logique de navigation.

J'ai réalisé le wireframe sur Canva. Il me fallait la version Desktop et Mobile.

Sur la homePage les carrés avec les croix représentent les images, les liens j'ai mis “Link” souligné, les traits foncés représentent les titres, les traits clairs du texte descriptif. Le système de connexion/inscription est représenté sur un rectangle gris qui au clique s'ouvre sous forme de modale.



Pour aperçu, une page d'animation :



Charte graphique



#000000

#1F2937

#374151

#075D2C

#FF7828

Police par défaut (Tailwind) : Inter

Concernant la charte graphique, je me suis inspirée de l'image du logo, généré par l'IA de Canva. J'ai récupéré les 2 couleurs principales pour écrire SURVIVAL en #075D2C avec la police : Zing Rust Base Rough et PARC en #FF7828 avec la police : Edo.

J'ai décidé de mettre mon header en noir sur une écriture blanche. Le menu apparaît en #FF7828 au survol.

Les cards sont en #374151. Et les background des pages sont en dégradé, du noir (pour rappel au header), vers le #1F2937



Les couleurs sombres sont un choix précis pour aller avec le thème du parc. Afin d'éviter que le site soit trop sombre, j'ai donc mis 2 couleurs vives afin de mettre en avant certains points comme le bouton connexion et inscription et rendre le site moins sinistre.

4. SPECIFICATIONS TECHNIQUES

A. Technologies Front-end

- React

React est l'une des bibliothèques les plus populaires pour le développement d'applications front-end. React est performant et flexible pour la création d'interfaces utilisateur dynamiques. Il permet de diviser l'interface en composants réutilisables, ce qui rend le code plus modulaire et maintenable. C'est également ce que l'on a appris durant la formation.

- Vite

Vite, est un outil qui rend le développement web plus rapide et plus fluide. Il charge seulement ce dont on a besoin à chaque instant, ce qui fait que la page se met à jour quand on modifie le code et que projet démarre presque instantanément.

- TypeScript

TypeScript ajoute un typage statique au langage. Cela permet de détecter les erreurs de type lors du développement, avant même d'exécuter le code. TypeScript aide à écrire du code plus robuste et moins sujet aux erreurs.

- Tailwind

Tailwind est un framework CSS qui permet de créer des interfaces utilisateur de manière rapide et modulaire, en utilisant des classes utilitaires.

- DaisyUI

DaisyUI est un plugin pour Tailwind qui fournit des composants prêts à l'emploi (comme des boutons, des cartes, des formulaires, etc.).

- 
- Axios

Axios est une bibliothèque JavaScript permettant de faire des requêtes HTTP. Elle simplifie l'envoi et la gestion des requêtes API et gère automatiquement les réponses. En plus, il fonctionne bien avec des fonctions qui attendent des résultats (appelées "asynchrones").

- react-router-dom

Cette bibliothèque permet de gérer la navigation entre les différentes pages d'une application React. Elle permet d'ajouter des routes dynamiques à l'application sans recharger la page, ce qui crée une expérience utilisateur fluide et sans interruption.

B. Technologies Back-end

- Node.js

Node.js est un environnement d'exécution JavaScript qui permet de faire tourner du code JavaScript côté serveur. Il a la capacité de traiter un grand nombre de requêtes simultanément.

- Express

Express est un framework minimaliste pour Node.js, utilisé pour faciliter la gestion des requêtes HTTP, la définition des routes et l'ajout de middlewares. Il simplifie le processus de création d'API, offrant un cadre de développement rapide et flexible.

- Sequelize

Sequelize est un ORM (Object-Relational Mapping) pour Node.js, qui permet d'interagir avec une base de données relationnelle SQL en utilisant des objets JavaScript. Cela simplifie la gestion des requêtes SQL complexes et permet de travailler avec des objets plutôt qu'avec des requêtes SQL brutes.

- Joi

Joi est une bibliothèque permettant de valider les données envoyées par l'utilisateur. Cela permet de vérifier que les informations soumises via les formulaires respectent les formats et les types attendus, et évite ainsi des erreurs côté serveur.



- CORS

CORS (Cross-Origin Resource Sharing) est un mécanisme qui permet de contrôler l'accès aux ressources d'une API depuis différents domaines. Ce middleware permet de définir des règles pour autoriser ou refuser l'accès aux API en fonction de l'origine des requêtes.

- Dotenv

Dotenv est une bibliothèque permettant de charger des variables d'environnement depuis un fichier .env. Cela permet de stocker des informations sensibles (comme des clés API ou des mots de passe) dans un fichier sécurisé, sans les exposer dans le code source.

- ESLint

ESLint est un outil de linting qui analyse le code JavaScript pour détecter des erreurs de syntaxe, des incohérences dans le style de code, et des problèmes potentiels. Cela permet de maintenir un code propre et lisible tout au long du développement.

- Jest

Jest est un framework de tests pour JavaScript. Il permet d'écrire et d'exécuter des tests unitaires et d'intégration pour s'assurer que le code fonctionne comme prévu.

- Supertest

Supertest est une bibliothèque qui permet de tester les API HTTP. Elle est souvent utilisée avec Jest pour tester les routes et les réponses du serveur, garantissant que les API fonctionnent correctement et renvoient les bonnes données.

- Base de données : PostgreSQL

PostgreSQL est une base de données relationnelle extrêmement robuste, avec des fonctionnalités avancées comme les transactions, les vues, et les types de données personnalisés. Elle est réputée pour sa stabilité, sa scalabilité et sa conformité aux normes SQL.



C. Sécurité de l'application

- Bcrypt

Bcrypt est une bibliothèque qui permet de hacher de manière sécurisée les mots de passe des utilisateurs. Son utilisation permet d'éviter les risques liés au stockage de mots de passe en clair et protège contre les attaques par force brute.

- JWT (JSON Web Token)

JWT permet de gérer l'authentification des utilisateurs en envoyant un token sécurisé au client. Ce token est utilisé pour valider les demandes de l'utilisateur sur le serveur sans avoir besoin de maintenir une session en continu. Cela rend l'authentification et l'autorisation plus sécurisées et performantes.

- Cookies

Un cookie est un fichier stocké sur le navigateur de l'utilisateur. Dans mon projet, le cookie est utilisé pour gérer la session de l'utilisateur. Lorsqu'un utilisateur se connecte, le serveur génère une session et envoie un identifiant unique au navigateur de l'utilisateur sous forme de cookie. La configuration du cookie garantit qu'il est sécurisé et protège contre les attaques XSS et CSRF.

5. ORGANISATION

A. Explication du choix d'un projet en autonomie

Initialement, le projet final de la formation "L'Apothéose" devait être réalisé en groupe. Ce projet avait pour objectif de synthétiser toutes les notions acquises au cours de la formation et de servir de support pour une présentation orale. Cependant, au cours de la réalisation, des défis organisationnels et des contraintes imprévues au sein du groupe ont impacté la répartition du travail et la progression du projet.

Ces circonstances m'ont amenée à prendre la décision de créer un projet individuel, intitulé Survival Parc, afin de garantir une production reflétant mon apprentissage et mes



compétences développées pendant la formation. Ce choix m'a permis de démontrer ma capacité à concevoir, coder et finaliser un projet de manière autonome, tout en respectant les exigences pédagogiques.

J'ai également effectué un stage de 2 mois chez Foodprint, une application permettant de calculer l'impact carbone des recettes dans des restaurations collectives. J'ai eu l'opportunité de participer activement au développement de la version 2 de cette application web en utilisant, notamment Next.js, et une base de données hébergée sur Azure.

Cette expérience m'a permis de renforcer mes compétences techniques dans des technologies complémentaires à celles utilisées dans mon projet individuel, ainsi que mes capacités de travail en équipe dans un environnement professionnel. Bien que je n'ai pas choisi d'utiliser ce projet de stage, car il ne pouvait pas valider une grande partie des compétences demandées, il reflète ma polyvalence et mon aptitude à m'adapter à des contextes et outils variés.

Survival Parc m'a donc offert l'opportunité de relever un nouveau défi personnel et de valoriser ma capacité d'adaptation face à une situation imprévue.

B. Méthodologie de travail

Étant seule sur ce projet, j'ai adopté une approche intuitive, basée sur une progression naturelle et logique des étapes de développement. Mon travail s'est déroulé en plusieurs phases clés, sans recourir à des méthodologies formelles ou à des outils spécifiques, mais en suivant un fil conducteur structuré :

- Rédaction du cahier des charges :

J'ai commencé par définir les besoins fonctionnels et techniques du projet. Cette étape m'a permis d'avoir une vision claire des objectifs à atteindre et des fonctionnalités à développer.

- Développement du back-end et de la base de données :

Une fois les besoins identifiés, j'ai construit le back-end et conçu la base de données en priorisant les aspects fondamentaux du projet, comme les fonctionnalités principales et la structure des données. Cependant la BDD a évolué au fil de la construction du projet, amenant à mettre à jour certaines fonctionnalités du back.



- Création du front-end :

Après avoir établi une base côté serveur, je me suis concentrée sur la partie front-end pour rendre l'application interactive et intuitive. Arrivée à cette partie du développement, j'ai constaté des erreurs ou des oubliés côté BDD. J'ai parfois dû re-questionner mon projet, ou modifier et mettre à jour ma BDD et mon Back afin de pouvoir maintenir ma ligne directive et rendre le front cohérent.

- Tests et ajustements :

Chaque étape du développement a été suivie de phases de tests pour vérifier la cohérence et la fonctionnalité des différentes parties du projet. J'ai globalement testé manuellement et grâce à Thunder Client.

Cette méthode flexible m'a permis d'avancer de manière fluide tout en ajustant mes priorités en fonction des besoins identifiés au fil du développement.

- Versioning

Le versioning consiste à garder une trace des différentes versions d'un projet ou d'un logiciel pendant son développement. Cela permet de savoir quand et pourquoi des modifications ont été faites sur le code ou les fichiers, et d'y revenir si nécessaire.

- Suivre les modifications : Chaque version du projet peut être identifiée de manière unique, ce qui permet de voir exactement quelles modifications ont été apportées, par qui, et à quel moment.
- Revenir à une version antérieure : En cas de problème ou d'erreur, le versioning permet de revenir facilement à une version précédente du projet qui était stable.
- Collaborer efficacement : Dans les équipes de développement, plusieurs personnes peuvent travailler sur différents aspects du même projet en même temps. Le versioning permet de fusionner ces modifications de manière ordonnée et sans conflit.

Comme versioning durant la formation, nous avons appris à utiliser Git. J'ai pu utiliser Git en équipe durant mon stage, cependant pour mon projet, étant donné que j'étais seule, il n'y a pas eu le versant collaboration d'équipe. J'ai utilisé Git pour suivre l'historique des fichiers, mais celui-ci peut-être également utilisé pour :

- 
- Création d'un dépôt Git : Au début du projet, un dépôt Git est créé pour suivre les modifications.
 - Commits : Chaque modification du code est enregistrée via un "commit". Un commit est un instantané de l'état des fichiers à un moment donné.
 - Branches : Git permet de travailler sur différentes branches du projet, ce qui permet de tester des fonctionnalités ou de corriger des bugs sans affecter la version principale (souvent appelée "master" ou "main").
 - Fusion (Merge) : Une fois les changements terminés, les différentes branches peuvent être fusionnées pour intégrer les modifications dans la version principale.

6. COMPÉTENCES DU REFERENTIELS

A. Front-end

CP 1 : Installer et configurer son environnement de travail

Pour le projet Survival Parc, j'ai mis en place un environnement de travail complet et adapté aux besoins du développement. Afin de séparer les différentes couches de l'application et faciliter la gestion du code, j'ai structuré le projet en deux dépôts Git distincts : l'un dédié au front et l'autre au back. Cette approche m'a permis de versionner chaque partie indépendamment et de maintenir une organisation claire tout au long du développement.

L'installation et la configuration des outils nécessaires ont été réalisées en fonction des technologies choisies pour répondre aux objectifs du projet, tout en optimisant la productivité et la fiabilité. Voici les principales étapes de configuration : mise en place des frameworks et bibliothèques, utilisation d'une base de données robuste, et préparation d'un environnement de développement sécurisé et performant.

Dans cette partie, je détaille les outils utilisés, leur installation et leur configuration, ainsi que l'organisation adoptée pour garantir un développement fluide et structuré.

- Les outils de développement nécessaires installés et configurés dans le back :
 - Node.js : Installation via le site officiel.
 - Express : Ajout avec **npm install express** pour créer une API rapide et efficace.

- 
- Sequelize : Installation via **npm install sequelize sequelize-cli** pour simplifier la gestion de la base de données et des migrations.
 - Bcrypt : Ajout avec **npm install bcrypt** pour hacher les mots de passe de manière sécurisée.
 - JWT (JSON Web Token) : Utilisé avec **npm install jsonwebtoken** pour gérer l'authentification.
 - Joi : Installation avec **npm install joi** pour valider les données des utilisateurs.
 - Cors : Ajout avec **npm install cors** pour configurer les permissions d'accès à l'API.
 - Dotenv : Ajout avec **npm install dotenv** pour gérer les variables d'environnement dans un fichier .env.
 - Nodemon : **npm install nodemon** pour automatiser le redémarrage du serveur à chaque modification du code.

- Les outils de développement nécessaires installés et configurés dans le front :
 - Vite : Crédit à la création du projet avec la commande **npm create vite@latest**, l'installation de Vite permet la configuration pour React et TypeScript.
 - Tailwind CSS et DaisyUI : Installation via npm et configuration du fichier **tailwind.config.js** pour inclure DaisyUI en tant que plugin.
 - Axios : Installation avec **npm install axios** pour gérer les appels API facilement.
 - React Router DOM : Ajout via **npm install react-router-dom** pour gérer la navigation entre les pages.
 - Framer Motion : est une bibliothèque JavaScript pour la création d'animations et d'interactions dans des applications React.
- Base de Données :
 - PostgreSQL : Installation locale et configuration avec Sequelize pour la connexion et les migrations. Je développerai cette partie dans CP : 5 Mettre en place en BDD relationnelle.
- Autres Outils :
 - ThunderClient : Utilisé pour tester les requêtes API rapidement, installé comme extension de VS Code.
 - Babel : Outil qui permet de convertir du JavaScript en une version compatible avec ma version de Node.js. pour les tests unitaires.



Pour certains composants front, je me suis aidée de <https://daisyui.com/>. Pour les icônes : <https://react-icons.github.io/react-icons/>

Globalement, pour des explications concrètes de l'ensemble du projet, je pouvais m'aider du site MDN : <https://developer.mozilla.org/fr/docs/Learn>

- Le système de veille permet de suivre les évolutions technologiques et les problématiques de sécurité.

Il s'agit de vérifier que les outils choisis pour le développement comme VSCode, GitHub, sont à jour et optimaux pour le projet.

Suivre les dernières versions des frameworks utilisés (comme Express, Sequelize, etc.) et voir si de nouvelles fonctionnalités ou améliorations sont disponibles.

Mise à jour régulière des dépendances via **npm audit** afin de détecter et corriger les vulnérabilités dans les bibliothèques utilisées.

CP 2 : Maquetter des interfaces utilisateur

Après l'élaboration du cahier des charges et des users stories j'ai pu réaliser un wireframe, une esquisse simple de l'interface pour montrer la disposition des éléments principaux (avant d'entrer dans le détail du design). J'ai réalisé la Homepage version desktop et mobile grâce à Canva. (Cf Annexe 7)

Pour la Homepage je voulais pouvoir mettre un aperçu du contenu du site, un carrousel des animations, et des photos des hôtels avec les liens de détails. Il fallait également mettre en fin de page, l'adresse et le lien de contact qui ouvre une fenêtre de boîte mail avec en destinataire l'adresse mail (fictive) du parc. J'ai souhaité créer un onglet de "connexion" et "inscription" qui au clique laisse apparaître une modale. A la connexion l'onglet change en "mon compte" et "déconnexion". Le header montre qu'on aura le logo du parc, et les liens vers les autres pages, ainsi qu'une icône de recherche et un panier qui apparaît uniquement à la connexion.

Pour le format mobile j'ai pris en référence le Samsung S8. Le format mobile propose un menu burger et un bouton "réserver" qui va directement sur la page des tickets.



Voici l'explication du menu burger :

```
● ● ●  
const [isMobileMenuOpen, setIsMobileMenuOpen] = useState(false);  
  
const toggleMobileMenu = () => {  
  setIsMobileMenuOpen(!isMobileMenuOpen);  
};  
  
const closeMobileMenu = () => {  
  setIsMobileMenuOpen(false);  
};
```

isMobileMenuOpen : un état booléen qui indique si le menu mobile est ouvert.

toggleMobileMenu : une fonction pour ouvrir/fermer le menu lorsqu'on clique sur le bouton.

closeMobileMenu : une fonction pour forcer la fermeture, utile lorsqu'un lien est cliqué.

```
● ● ●  
<button className="md:hidden text-2xl z-50" onClick={toggleMobileMenu}>  
  {isMobileMenuOpen ? <FiX /> : <FiMenu />}  
</button>
```

Ce bouton est visible uniquement sur les petits écrans grâce à la classe *md:hidden*. L'icône change dynamiquement : *<FiX />* pour une croix (fermer). *<FiMenu />* pour un menu burger (ouvrir).

onClick={toggleMobileMenu} appelle la fonction pour basculer entre ouvert et fermé.

```
● ● ●  
{isMobileMenuOpen && (  
  <motion.nav  
    initial={{ x: '-100%' }}  
    animate={{ x: 0 }}  
    exit={{ x: '-100%' }}  
    className="bg-black text-white fixed inset-0 p-6 mt-20 md:hidden z-  
40">  
    <div className="flex flex-col space-y-6">  
      <MobileMenu closeMenu={closeMobileMenu} />  
    </div>  
  </motion.nav>  
)}
```

Condition *isMobileMenuOpen* : Le menu est affiché uniquement si *isMobileMenuOpen* est true.

Animations avec motion.nav et CSS avec la Classe Tailwind

Suite au wireframe, j'ai pu réaliser la maquette en m'aidant de la charte graphique que j'avais réalisée dans le cahier des charges. (cf Annexe 8).



CP 3 : Réaliser des interfaces utilisateur statiques

Ici l'objectif est de concevoir et réaliser une page web statique, en intégrant une interface responsive qui s'adapte à différents types d'écrans.

```
/* @type {import('tailwindcss').Config} */
export default {
  content: [
    './index.html',
    './src/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {
      screens: {
        sm: '640px',
        md: '768px',
        lg: '1024px',
        xl: '1280px',
        '2xl': '1536px',
      },
    },
    plugins: [require("daisyui")],
  }
}
```

Voici la configuration utilisée pour Tailwind CSS dans le fichier *tailwind.config.js*. Elle définit les breakpoints pour rendre le site responsive et spécifie les fichiers scannés pour les classes CSS utilisées dans le design. Un plugin supplémentaire, DaisyUI, est aussi intégré pour des composants stylisés préconçus.

Dans mes pages, j'ai ajouté des styles afin que les pages soient responsives.

sm:, md:, lg:, xl: : Ces préfixes indiquent des styles spécifiques pour des tailles d'écran différentes.

Cf Annexe 9 : extrait du code. En rouge le code expliqué)

- *sm:w-[70px]* : largeur pour les écrans $\geq 640\text{px}$.
- *w-[150px]* : largeur par défaut pour les petits écrans.
- *md:w-[100px], lg:w-[200px], xl:w-[250px]* : largeurs adaptées pour les écrans $\geq 768\text{px}, 1024\text{px}$, et 1280px respectivement.
- *className="grid-cols-1 sm:grid-cols-2 lg:grid-cols-3"* : La grille change dynamiquement de 1 à 3 colonnes selon la taille de l'écran.
- *pt-20 sm:pt-40* : Padding en haut ajusté selon les breakpoints
- *space-y-12 sm:space-y-16* : Espace vertical dynamique entre les sections pour améliorer l'esthétique selon la taille de l'écran.
- *text-3xl sm:text-5xl* : Taille de police du titre augmentée sur les écrans moyens et grands pour une meilleure lisibilité.

- 
- *w-full h-auto* : Les images remplissent la largeur disponible et conservent leurs proportions
 - *flex flex-col item-center* : Flexibilité dans la disposition. La disposition en colonne (flex-col) est idéale pour les petits écrans, car elle empile les éléments verticalement.

Grâce à la configuration de Tailwind CSS et ses classes utilitaires, la majorité des adaptations responsives sont gérées efficacement sans nécessiter de styles personnalisés complexes. Cela permet d'avoir un code maintenable et cohérent sur toutes les pages du site. (Aperçu des pages responsives Cf Annexe 9 Bis).

CP 4 : Développer la partie dynamique des interfaces utilisateur web ou web mobile

Dynamiser une interface utilisateur a pour objectif d'améliorer l'expérience utilisateur en la rendant plus interactive, fluide et réactive. Une interface dynamique permet de répondre rapidement aux actions de l'utilisateur sans avoir à recharger la page, ce qui rend l'application plus agréable à utiliser et plus efficace. Ici je vais expliquer des mécanismes d'événements côté client et démontrer la capacité à gérer l'interface d'une application avec React.

A titre d'exemple, sur toutes les pages du site, il y a un onglet de connexion/inscription. Au clique une modale s'ouvre.

- **Gestion de l'ouverture et fermeture des modales connexion : Gestion des événements côté client avec useState.**

useState permet la gestion de l'état en local. Permet de stocker et de mettre à jour des informations sans recharger la page.



```
● ● ●
{!isAuthenticated ? (
  <button onClick={openLoginModal}>Connexion</button>
  <button onClick={openSignupModal}>Inscription</button>
) : (
  <button onClick={handleLogout}>Déconnexion</button>
  <button onClick={handleGoToProfile}>Mon compte</button>
)}
```

L'ouverture et la fermeture des modales sont gérées par l'utilisation de l'état local dans React avec le hook useState. Dans ce cas, deux variables d'état sont utilisées pour savoir si les modales de connexion ou d'inscription doivent être ouvertes ou non. Par défaut, ces deux modales sont fermées, car leurs états sont initialisés à false.

Lorsque l'utilisateur clique sur le bouton de connexion, un événement `onClick` est déclenché. Si l'utilisateur est connecté, des boutons de déconnexion et de profil sont affichés. Si l'utilisateur n'est pas connecté, des boutons de connexion et d'inscription sont visibles.

```
● ● ●
{isLoginModalOpen && (
  <LoginModal
    isOpen={isLoginModalOpen}
    onClose={closeLoginModal}
    onLoginSuccess={handleLoginSuccess}
    setLoginCredentials={setLoginCredentials}
  />
)}
```

Cet événement permet d'afficher le composant enfant `LoginModal` qui reçoit plusieurs props lorsque `isLoginModalOpen` est vrai. Les fonctions suivantes sont appelées en fonction des événements spécifiques déclenchés par l'utilisateur

comme `isOpen`, `onClose` (qui fermera la modale) et `setLoginCredentials` pour gérer les informations du formulaire de connexion.

- **Gestion conditionnelle de l'interface utilisateur avec l'état d'authentification : Utilisation d'événements asynchrones (promesses) avec then/catch**

Lorsqu'un utilisateur tente de se connecter ou de s'inscrire, les fonctions sont asynchrones et renvoient des promesses. Le frontend gère ces promesses, déclenche l'interface appropriée (par exemple, fermer la modale ou afficher un message d'erreur).

- `loginUser(email, password)` : Essaye de connecter l'utilisateur avec les informations fournies.
- `.then()` : Exécuté si la connexion réussit, on ferme la modale.
- `.catch()` : Exécuté si une erreur survient, l'erreur est affichée dans la console.



J'ai utilisé plusieurs actions rendant mon application dynamique telles que :

- Gestion des formulaires : Permet de dynamiser des formulaires avec des champs qui apparaissent, disparaissent ou sont validés en temps réel sans recharger la page.
- Filtrage dynamique : Permet de filtrer une liste d'éléments affichée sur la page.

C. Back

CP 5 : Mettre en place une base de données relationnelle

Dans mon projet j'ai conçu une base de données en utilisant PostgreSQL. J'ai d'abord utilisé des outils comme le MCD et le MLD vu précédemment pour définir la structure de la base de données, ce qui a permis de répondre aux besoins fonctionnels du projet.

La base de données a été structurée autour des tables principales : User, Profile, Reservation, Hotel, et Animation. J'ai utilisé des commandes SQL pour la création et la gestion des tables, ainsi que des scripts automatisés pour faciliter cette gestion.

Voici les étapes de la mise en place :

- Installation et configuration de PostgreSQL

Installation de PostgreSQL via son site officiel :

- Création de la base de données et des utilisateurs
- Connexion à la BDD grâce à la commande `psql -U postgres` et `\c survival`
- Création de la BDD : `CREATE DATABASE survival;`
- Création d'un utilisateur et d'un mot de passe : `CREATE USER user WITH PASSWORD 'survival';`
- Pour donner à cet utilisateur les droits nécessaires sur la base de données projet : `GRANT ALL PRIVILEGES ON DATABASE survival TO user;`

- Configuration de Sequelize pour la connexion :

Dans le fichier *dbclient.js*, j'ai configuré Sequelize pour connecter l'application à la base de données PostgreSQL. Les informations de connexion sont stockées dans le fichier *.env* afin de sécuriser les données sensibles :

```
import { Sequelize } from "sequelize";
import dotenv from "dotenv";

dotenv.config();

export const sequelize = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASSWORD,
  {
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    dialect: process.env.DB_DIALECT,
  }
);
```

- DB_USER : Nom de l'utilisateur PostgreSQL (survival)
- DB_PASSWORD : Mot de passe de l'utilisateur
- DB_NAME : Nom de la base de données
- DB_HOST : Adresse du serveur de base de données
- DB_PORT : Port PostgreSQL (par défaut 5432)
- DB_DIALECT : Dialecte SQL (postgres)

- Gestion des scripts dans *package.json* :

```
"db:reset": "node src/seeders/resetDatabase.js",
"db:create": "node src/seeders/createTable.js",
"db:seed": "node src/seeders/populateTable.js",
```

Commandes définies pour faciliter la gestion de la base de données :

- *npm run db:create* : Crée les tables.
- *npm run db:seed* : Insère des données dans les tables.
- *npm run db:reset* : Supprime et recrée les tables (utilisé principalement pour le développement).

- Pour vérifier la création des tables :

- Lister les tables : *\dt*
- Afficher la structure d'une table : *\d animations*

```
sequelize
.authenticate()
.then(() => {
  console.log(
    "✅ Connection to the database has been established successfully."
);
```

Dans *index.js* je vérifie la connexion à la BDD avec Sequelize, cela garantit que l'application peut interagir correctement avec PostgreSQL avant de démarrer le serveur.



CP 6 : Développer des composants d'accès aux données SQL

- Création des modèles Sequelize pour représenter les tables de la base de données :

Les modèles dans un ORM comme Sequelize sont des représentations de tables de la BDD. Chaque modèle définit la structure d'une table (champs, types de données) et les relations entre différentes tables. Ils permettent d'interagir avec la BDD en utilisant des méthodes plutôt que d'écrire des requêtes SQL directement, facilitant ainsi la gestion des données.

(Exemple de model pour l'hôtel : Cf Annexe 10)

J'ai créé un fichier pour les associations et l'initialisation des models. Nous retrouvons les associations telles que hasOne,hasMany, belongsToMany.

```
// Initialisation des modèles et associations
export const initializeModels = () => {
  const models = [
    User: User.init(sequelize),
    Profile: Profile.init(sequelize),
    Reservation: Reservation.init(sequelize),
    Hotel: Hotel.init(sequelize),
    Animation: Animation.init(sequelize),
    ProfileHotel: ProfileHotel.init(sequelize),
  ];

  // Associations entre les modèles
  UserhasOne(Profile, { foreignKey: "userId", sourceKey: "id" });
  Profile.belongsTo(User, { foreignKey: "userId", targetKey: "id" });

  ProfilehasMany(Reservation, { foreignKey: "profileId", sourceKey: "id" });
  Reservation.belongsTo(Profile, { foreignKey: "profileId", targetKey: "id" });

  Profile.belongsToMany(Hotel, {
    through: ProfileHotel,
    foreignKey: "profileId",
    otherKey: "hotelId",
    as: "profileHotels",
  });

  Hotel.belongsToMany(Profile, {
    through: ProfileHotel,
    foreignKey: "hotelId",
    otherKey: "profileId",
    as: "hotelProfiles",
  });

  return models;
};
```

- Interaction avec les données via Sequelize :

Dans cette partie, j'explique comment j'utilise les méthodes de Sequelize pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur la BDD. Voici quelques exemples d'utilisation des principales méthodes Sequelize :

- findAll() : Pour récupérer plusieurs enregistrements.
- findOne() ou findByPk() : Pour récupérer un seul enregistrement.
- create() : Pour créer un nouvel enregistrement.
- update() : Pour mettre à jour un enregistrement existant.
- destroy() : Pour supprimer un enregistrement.

Exemple de récupération de toutes les animations :

```
const animations = await Animation.findAll();
```

- Gestion des données dans le modèle MVC (Model-View-Controller) :

Dans ce modèle les contrôleurs sont responsables des requêtes HTTP, de l'appel aux modèles pour récupérer ou manipuler les données, et de l'envoi des réponses.

Exemple : récupération d'un user par son id :

```
export const getUserId = ctrlWrapper(async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    attributes: ["id", "email", "role", "createdAt", "updatedAt"],
  });
  if (!user) return error404(res, "User not found");

  successResponse(res, "User fetched successfully", user);
});
```

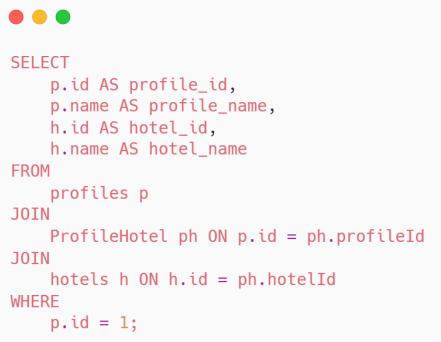
- `getUserId` : Il s'agit d'une fonction exportée qui représente le contrôleur pour récupérer un utilisateur à partir de son ID.
- `ctrlWrapper` : Cette fonction est utilisée pour envelopper le contrôleur asynchrone. Cela permet de gérer les erreurs de manière centralisée, souvent en capturant les erreurs non gérées dans une promesse ou dans des fonctions asynchrones (comme `await`).
- `User.findById` : Méthode pour rechercher un utilisateur par son ID primaire (Primary Key). Cela signifie que l'ID de l'utilisateur sera extrait des paramètres de la requête (`req.params.id`).
- `req.params.id` : Les paramètres de la requête sont des données envoyées dans l'URL de la requête HTTP. Par exemple, si la route est `/user/:id`, alors `req.params.id` va contenir la valeur de `id` dans l'URL `/user/3` ici `req.params.id` serait 3.
- `attributes` : Ici, je définis quels champs de l'utilisateur je souhaite retourner dans la réponse JSON.
- `if (!user)` : Cette ligne vérifie si l'utilisateur n'a pas été trouvé dans la BDD. Si `findById` ne trouve aucun utilisateur avec l'ID spécifié, il retourne null, et donc la condition `if (!user)` sera vraie.
- `return error404(res, "User not found");` : Si l'utilisateur n'existe pas, la fonction `error404` est appelée pour renvoyer une réponse d'erreur 404. Elle prend comme argument la réponse (`res`) et le message "User not found" qui sera envoyé dans la réponse JSON.
- `successResponse` : Si l'utilisateur est trouvé, la fonction `successResponse` est appelée pour renvoyer une réponse réussie. Elle prend trois arguments :
 - ◆ `res` : L'objet de réponse qui permet d'envoyer des données au client.
 - ◆ `"User fetched successfully"` : Un message décrivant le succès de l'opération.
 - ◆ `user` : L'utilisateur trouvé, qui sera envoyé en réponse sous forme de données JSON.

- Exemples de requêtes SQL via Sequelize :

`SELECT * FROM User;` : sélectionner tous les utilisateurs de la table User

`DROP TABLE animations CASCADE;` : supprimer la table animations

Nous pouvons faire des requêtes plus complexes telles que récupérer un profil et les hôtels qu'il a réservés en utilisant la table d'association `ProfileHotel`.



```

SELECT
    p.id AS profile_id,
    p.name AS profile_name,
    h.id AS hotel_id,
    h.name AS hotel_name
FROM
    profiles p
JOIN
    ProfileHotel ph ON p.id = ph.profileId
JOIN
    hotels h ON h.id = ph.hotelId
WHERE
    p.id = 1;

```

- ***SELECT*** : On sélectionne les informations que l'on veut récupérer :
- ***p.id*** : L'ID du profil.
- ***p.name*** : Le nom du profil.
- ***h.id*** : L'ID de l'hôtel réservé.
- ***h.name*** : Le nom de l'hôtel.
- ***FROM profiles p*** : La requête commence avec la table profiles (p), qui contient les informations des profils.
- ***JOIN ProfileHotel ph ON p.id = ph.profileId*** : On fait une jointure avec la table d'association ProfileHotel (ph) où profileId correspond à p.id. Cela permet de relier chaque profil aux hôtels qu'il a réservés.
- ***JOIN hotels h ON h.id = ph.hotelId*** : Une deuxième jointure avec la table hotels (h) pour récupérer les informations des hôtels associés à chaque profil via la table d'association.
- ***WHERE p.id = 1*** : On filtre les résultats pour ne récupérer que les hôtels réservés par un profil avec l'ID 1.

Résultat :

profile_id	profile_name	hotel_id	hotel_name
1	Joel Miller	1	Le refuge
1	Joel Miller	2	Post-apocalypse

CP 7 : Développer des composants métier côté serveur

- Architecture

Dans ce projet, le développement des composants métier côté serveur repose sur une architecture, structurée selon le modèle MVC (Model-View-Controller). Ce modèle permet de séparer clairement les responsabilités de chaque couche : les Modèles (gestion des données et



logique métier), les Contrôleurs (traitement des requêtes HTTP et communication avec les modèles), et les réponses envoyées au client sous forme JSON. Ce choix architectural garantit une meilleure lisibilité du code, une évolutivité accrue et une maintenance simplifiée.

La Programmation Orientée Objet (POO) est une manière d'écrire du code où l'on utilise des objets pour représenter des éléments ou des concepts. Un objet a deux choses principales :

Des propriétés (ou attributs), qui sont des informations sur l'objet. Par exemple, un objet "Utilisateur" peut avoir des propriétés comme "nom", "email", "mot de passe".

Des comportements (ou méthodes), qui sont des actions que l'objet peut effectuer. Par exemple, un objet "Utilisateur" pourrait avoir une méthode pour se connecter ou mettre à jour son profil.

Dans mon projet, la POO est utilisée pour représenter des entités comme utilisateurs, profils, hôtels, et réservations. Cela permet de mieux organiser le code et de réutiliser des actions (méthodes) pour travailler avec ces entités facilement, comme par exemple créer un utilisateur ou modifier une réservation. (Cf Annexe 11)

- La circulation des données :



```
router.post("/register", validateRegister, cw(userController.createUser));  
router.get("/", cw(userController.getAllUsers));  
router.get("/:id", cw(userController.getUserById));  
router.put("/:id", cw(userController.updateUser));  
router.delete("/:id", cw(userController.deleteUser));
```

Les requêtes HTTP envoyées par le client arrivent dans l'api à travers les routes définies. Ces routes définissent les endpoints disponibles (ex. : /user), le type de requête HTTP (GET, POST, PUT, DELETE) et associent chaque requête à un contrôleur spécifique.

- Traitement par les contrôleurs :

Une fois qu'une route identifie et redirige une requête, elle est prise en charge par le contrôleur correspondant. Les contrôleurs agissent comme des intermédiaires entre la requête du client et la BDD. Ils reçoivent la requête et extraient les données nécessaires (depuis les paramètres, le corps de la requête ou l'URL).



- Interaction avec les modèles :

Les contrôleurs s'appuient sur les modèles, qui encapsulent la logique métier. Ces modèles interagissent directement avec la BDD via Sequelize.

Retour des données au client :

Une fois que les données nécessaires sont récupérées ou manipulées, les contrôleurs retournent une réponse structurée en JSON au client.

- Rôle des middlewares :

Avant que les requêtes atteignent les contrôleurs, elles peuvent passer par des middlewares. Ces derniers servent à Gérer la sécurité (par ex., validation des tokens d'authentification, validate des données via Joi).

- La sécurité

La sécurité a été une priorité dans la mise en œuvre des composants serveur. Cela inclut la gestion des accès via un middleware pour la vérification des tokens JWT, la validation des données entrantes pour prévenir les injections SQL ou d'autres attaques, ainsi que la sécurisation des informations sensibles grâce à l'utilisation de variables d'environnement et du chiffrement des mots de passe.

- .env :

Le fichier .env est utilisé pour stocker des informations sensibles :

- ➔ La clé secrète utilisée pour JSON Web Tokens (JWT) (JWT_SECRET).
- ➔ Les informations de connexion à la BDD (nom d'utilisateur, mot de passe, nom de la base, etc.).

En gardant ces informations hors du code source, elles ne sont pas exposées dans le dépôt de versionnement (grâce à l'utilisation d'un fichier .gitignore).

- Middleware d'authentification par token :

Le middleware de token (verifyToken.js) s'intègre dans l'architecture en interceptant les requêtes avant qu'elles n'atteignent les contrôleurs ou les autres parties du back. Dans une architecture MVC, ce middleware fonctionne comme une couche de sécurité et



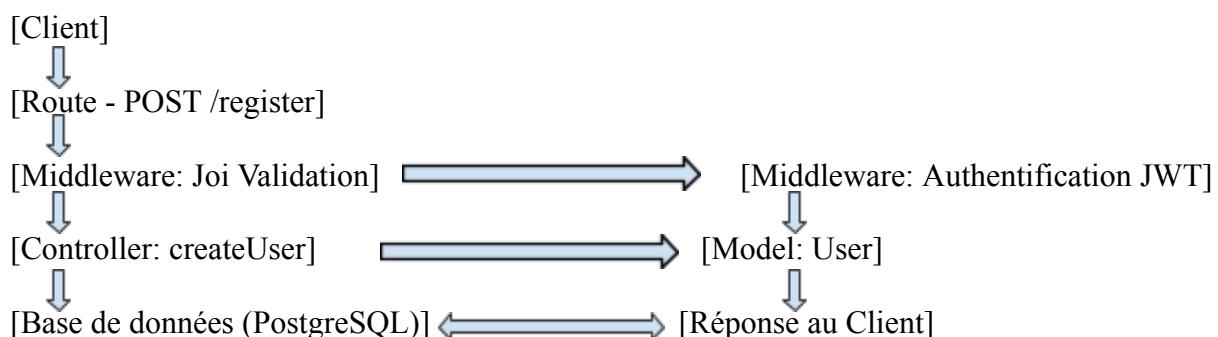
d'authentification entre le client et les contrôleurs. Il vérifie si l'utilisateur a un jeton valide, en s'assurant que la requête provient bien d'un utilisateur authentifié.

- Configuration de CORS (Cross-Origin Resource Sharing) dans index.js :

```
// Middlewares Cors
app.use(
  cors({
    origin: "http://localhost:5173",
    credentials: true,
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  })
);
```

- Origine autorisée (origin) : Le domaine client (<http://localhost:5173>) est explicitement autorisé à faire des requêtes vers l'api. Cela empêche les requêtes non autorisées provenant d'autres domaines (protection contre les attaques Cross-Site Request Forgery - CSRF).
- Autorisation des cookies (credentials) : La propriété credentials: true autorise l'échange de cookies entre le front et le back. Cela permet de stocker les tokens ou sessions côté client tout en les envoyant automatiquement avec chaque requête.
- Méthodes autorisées (methods) : Seules les requêtes utilisant les méthodes GET, POST, PUT, et DELETE seront acceptées.
- Headers autorisés (allowedHeaders) : Les headers spécifiques (comme Content-Type et Authorization) sont autorisés, ce qui est utile pour les requêtes nécessitant des tokens d'authentification ou des données JSON.

Schéma de circulation des données :





CP 8 : Documenter le déploiement d'une application dynamique

La mise en production est un processus complet qui permet de rendre une application opérationnelle, sécurisée et accessible aux utilisateurs finaux après qu'elle ait été développée et testée.

- Récupérer le projet et l'installer :

- Récupération du code source : Un développeur peut récupérer le projet depuis un système de gestion de version, ici ça serait Git, en clonant les repository via `git clone` dans 2 terminaux séparés.
- Installer les dépendances : Il faudra ensuite installer toutes les dépendances nécessaires pour faire fonctionner le projet via `npm install`
- Configurer les variables d'environnement : L'application devra être configurée pour l'environnement de production, ce qui inclut la gestion des variables d'environnement (comme les clés API, la base de données, les secrets, etc.). Cela se fait à travers le fichier `.env`. Pour cela on informe qu'il faut copier le fichier `.env.example`.
- lancer le serveur : Mode développement `npm run dev`

Mode production front, `cd client` puis `npm run build` (pour créer le bundle front) et `npm run preview` (pour lancer le front en mode production)

Mode production back, `cd api` puis `npm run start` (pour lancer le back en mode prod)

- Vérifications avant la mise en production :

- Avant de déployer en production, le projet doit être testé pour s'assurer qu'il fonctionne correctement. Cela peut inclure des tests unitaires ou des tests d'intégration. (Cf JEU D'ESSAI)

- Le processus de mise en production :

Déploiement sur le serveur de production : Une fois toutes les configurations effectuées, on déploie le code sur un serveur de production par exemple CI/CD avec GitHub Actions. Je n'ai pas effectué cette démarche dans mon projet, mais voici ce qu'il faudrait faire pour le déployer sur ce serveur de production.



Lorsqu'on commit du code dans le dépôt GitHub, un pipeline CI (Continuous Integration / Intégration Continue)/CD (Continuous Delivery ou Continuous Deployment / Livraison ou Déploiement Continu) via GitHub Actions peut automatiquement tester, construire et déployer l'application.

Il faut créer un fichier `.github/workflows/deploy.yml` où l'on définit un ensemble de règles pour que GitHub prenne en charge le déploiement du projet sur le serveur de production. Cela permettrait de déployer automatiquement les mises à jour dès qu'elles sont poussées sur la branche principale du dépôt.

- **La veille technologique :**

L'objectif ici est d'assurer une mise en production stable, sécurisée et optimisée. Cela comprend la mise à jour des outils utilisés pour le déploiement, la surveillance de l'application en production, et la gestion des risques de sécurité une fois l'application déployée.

Il faut suivre les évolutions des outils d'automatisation du déploiement comme GitHub Actions. Il faut aussi se concentrer sur la gestion des risques de sécurité comme la gestion des secrets (comme les clés API et les tokens), les mises à jour de sécurité des serveurs.

7. ELEMENTS DE SECURITE

A. Authentification sécurisée (JWT TOKEN)

L'authentification sécurisée avec les tokens JWT (JSON Web Tokens) est une méthode pour gérer les connexions des utilisateurs. Les JWT sont des jetons contenant des informations codées (comme l'identité de l'utilisateur). Ils permettent de valider l'identité d'un utilisateur sans avoir à stocker son état sur le localstorage.

- Fichier `.env` - Configuration des variables d'environnement :

Tout d'abord, il est important de stocker certaines valeurs sensibles comme la clé secrète (`JWT_SECRET`) et la durée d'expiration du token (`JWT_EXPIRES_IN`) dans un fichier `.env` pour des raisons de sécurité et pour garder ces valeurs flexibles.

- Fichier *jwt.js* - Génération et Vérification du Token :

```

● ● ●

import jwt from "jsonwebtoken";
import dotenv from "dotenv";

dotenv.config();

// Fonction pour générer un token JWT
export const generateToken = (user) => {
  if (!user || !user.id || !user.role) {
    throw new Error("User object must contain id and role");
  }

  return jwt.sign(
    { id: user.id, role: user.role }, // Payload
    process.env.JWT_SECRET, // Clé secrète
    { expiresIn: process.env.JWT_EXPIRES_IN || "1h" } // Durée d'expiration
  );
};

// Fonction pour vérifier un token JWT
export const verifyToken = (token) => {
  try {
    return jwt.verify(token, process.env.JWT_SECRET); // Vérifie et retourne le payload
  } catch (error) {
    throw new Error("Invalid token");
  }
};

```

generateToken(user) :

Cette fonction crée un token JWT pour un utilisateur donné. Elle prend un objet user contenant l'id et le rôle, qui seront inclus dans le payload (ce qui contient les données) du token.

Le token est ensuite signé avec la clé secrète (*process.env.JWT_SECRET*), et on reprend la durée d'expiration définie dans .env.

verifyToken(token) : Cette fonction prend un token et le vérifie en utilisant la même clé secrète (*process.env.JWT_SECRET*). Si le token est valide et n'a pas expiré, elle retourne le payload. Si le token est invalide ou expiré, une erreur est lancée.

- Middleware d'authentification : *authenticateToken* :

Le middleware *authenticateToken* sera utilisé pour vérifier si un utilisateur est authentifié avant d'accéder à certaines routes protégées comme réservation, profil.

```

● ● ●

import { verifyToken } from "../utils/jwt.js"; // Import de la fonction depuis utils
import { unauthorizedResponse } from "../middlewares/errorResponse.js";

export const authenticateToken = (req, res, next) => {
  const token = req.cookies.accessToken;

  if (!token) {
    return unauthorizedResponse(res, "Access token missing");
  }

  try {
    // Utilise la fonction générique verifyToken pour valider le token
    const decoded = verifyToken(token);
    req.userId = decoded.id; // L'ID de l'utilisateur (dans le payload du token)
    req.userRole = decoded.role; // Le rôle de l'utilisateur

    next(); // Passe au middleware suivant
  } catch (error) {
    return unauthorizedResponse(res, "Invalid or expired token");
  }
};

```

authenticateToken :

On récupère le token d'accès stocké dans les cookies (*req.cookies.accessToken*).

Si le token est présent, il appelle *verifyToken(token)* pour le valider.



Si le token est valide, il extrait l'ID et le rôle de l'utilisateur à partir du payload du token et les ajoute à l'objet req pour les rendre accessibles dans les autres middlewares ou routes. Si le token est manquant ou invalide, il renvoie une réponse 401 Unauthorized.

- Controller authController.js - Gestion de l'authentification : (Cf Annexe 12)

Le contrôleur contient la logique pour le login et le logout.

login : Vérifie les informations d'identification de l'utilisateur. Si elles sont valides, il génère un token JWT avec la fonction generateToken. Le token est ensuite envoyé dans un cookie au client, et l'utilisateur est considéré comme connecté.

logout : Lorsque l'utilisateur se déconnecte, son token est supprimé de la base de données et du cookie.

B. Contrôle de champs (Joi) (Cf Annexe 13)

Joi est une bibliothèque JavaScript qui permet le contrôle de champs afin d'assurer que les données entrantes respectent les règles attendues (formats, limite de caractère...). Grâce à cela, on peut prévenir des erreurs, protéger contre des attaques comme l'injection, et garantir que l'application ne traite que des données conformes et sécurisées.

Joi est utilisé dans un middleware (ici validateLogin). Ce middleware est appliqué dans le router avant d'atteindre les contrôleurs, ce qui permet de centraliser et de simplifier la validation des données. Une fois les données validées, le contrôleur peut exécuter la logique métier.

- *Joi.object* : défini un schéma pour les données de la requête (email et password).
- *.messages* : personnalise les messages d'erreur pour chaque règle de validation.
- *schema.validate(req.body)* : valide le corps de la requête avec ce schéma.
- *badRequestResponse(res, error.details[0].message)* : Si la validation échoue, ça renvoie un message d'erreur via la fonction *badRequestResponse*.

- Utilisation du middleware dans le router (authRouter.js) :

Ensuite, une fois le middleware de validation créé, il faut l'utiliser dans le router pour valider les données de connexion avant qu'elles n'arrivent dans le contrôleur login.



```
router.post("/login", validateLogin, login);
```

Lorsque la route /login est appelée, validateLogin est exécuté avant le contrôleur login. Si la validation échoue, la requête sera interrompue et un message d'erreur sera renvoyé. Si la validation réussit, le contrôleur login sera exécuté.

C. Gérer les sessions et renforcer la sécurité contre XSS et CSRF

La gestion des sessions repose souvent sur les cookies, un moyen de stocker des informations côté client tout en maintenant une session utilisateur persistante, afin d'éviter de se reconnecter à chaque page. Cela est géré avec des sessions qui utilisent des cookies pour stocker des infos sur l'utilisateur sur son navigateur. Pour éviter que quelqu'un exploite ces sessions pour attaquer l'application, on renforce la sécurité contre deux types d'attaques :

XSS (Cross-Site Scripting) : C'est quand un hacker injecte du code malveillant (comme un script) dans une page que l'utilisateur va ouvrir. Par exemple : une fausse alerte ou un champ de formulaire qui vole des données.

CSRF (Cross-Site Request Forgery) : C'est quand un hacker pousse un utilisateur déjà connecté à faire une action qu'il n'a pas voulu, comme transférer de l'argent. Par exemple : le hacker cache un lien malveillant dans un email ou un site externe. Pour éviter cela on peut utiliser des jetons de sécurité (tokens) pour valider chaque action importante.

- Fonctionnement des cookies :

```
// Fonction pour définir le cookie
export const setCookies = (res, accessToken) => {
  res.cookie("accessToken", accessToken, {
    httpOnly: true, // Empêche l'accès au cookie via JavaScript
    secure: process.env.NODE_ENV === "production", // Utiliser le cookie sécurisé en production
    maxAge: parseInt(process.env.COOKIE_MAX_AGE) || 3600000, // Durée de vie du cookie
    sameSite: "lax", // Définit la stratégie de partage entre sites (peut être 'strict' ou 'none')
  });
};

// Fonction pour effacer le cookie
export const clearCookies = (res) => {
  res.clearCookie("accessToken", {
    httpOnly: true,
    secure: process.env.NODE_ENV === "production",
    sameSite: "lax",
  });
};
```



Le fichier `cookieUtils.js` contient les fonctions qui gèrent les cookies, c'est-à-dire la création et la suppression des cookies contenant le token JWT.

- `httpOnly` : Empêche le cookie d'être accessible côté client, ce qui le protège des attaques XSS.
- `secure` : Garantit que le cookie est envoyé uniquement sur une connexion HTTPS.
- `maxAge` : Définit la durée de vie du cookie. Cela correspond à la durée d'expiration du token JWT.
- `sameSite` : Empêche l'envoi de cookies dans des requêtes provenant d'autres sites (protection contre les attaques CSRF).
- `clearCookies` : supprime le cookie `accessToken` du navigateur en utilisant les mêmes options de sécurité que celles définies lors de la création du cookie.

Dans `index.js`, il y a des middlewares configurés pour gérer les cookies et les sessions. Ces middlewares sont utilisés pour assurer que les cookies sont traités correctement tout au long du cycle de la requête HTTP.

Le middleware cookie-parser `app.use(cookieParser())`; permet de lire le cookie envoyé par le client (le cookie contenant le token JWT) et de le rendre disponible dans `req.cookies`.

Le middleware CORS permet les requêtes cross-origin, les cookies doivent être envoyés en `credentials: true`, ça permet au cookie contenant le token JWT d'être envoyé avec les requêtes au serveur.

- Connexion (login) :

Lorsqu'un utilisateur se connecte avec ses informations d'identification (email et mot de passe), un token JWT est généré. Ensuite, ce token est stocké dans un cookie grâce à la fonction `setCookies. setCookies(res, token)`;

- Déconnexion (logout) :

Lors de la déconnexion, le cookie contenant le token JWT est supprimé grâce à la fonction `clearCookies(res);`



D. Hashage du mot de passe

Bcrypt est une bibliothèque utilisée pour hacher les mots de passe de manière sécurisée. Elle est l'une des solutions pour protéger les mots de passe avant de les stocker dans une base de données.

Bcrypt utilise un mécanisme appelé *salage*. Cela signifie qu'il ajoute une donnée aléatoire (appelée "sel") au mot de passe avant de le hasher. Cela rend le mot de passe hashé unique, même si deux utilisateurs ont le même mot de passe. Il permet également de vérifier un mot de passe en le comparant au hash enregistré dans la base de données.

Dans mon projet, le hashage intervient à deux moments :

- Lors de la création d'un utilisateur : le mot de passe est hashé avant d'être sauvegardé dans la base de données.
- Lors de la connexion d'un utilisateur : le mot de passe hashé est comparé avec celui stocké dans la base de données pour vérifier sa validité.

- Dans le fichier userController.js pour la création d'un utilisateur :

```
const hashedPassword = await bcrypt.hash(password, 10);
```

Le mot de passe est hashé avec la fonction `bcrypt.hash(password, 10)`. Le nombre 10 correspond au nombre de salages (*salt rounds*), ce qui augmente la complexité. Le mot de passe hashé est ensuite stocké dans la base de données au lieu du mot de passe brut.

- Dans le fichier authController.js pour la connexion d'un utilisateur :

```
const isPasswordValid = await bcrypt.compare(password, user.password);
if (!isPasswordValid) {
  return badRequestResponse(res, "Erreur de mot de passe");
}
```

La fonction `bcrypt.compare(password, user.password)` vérifie si le mot de passe brut correspond à son empreinte. Si les mots de passe ne correspondent pas, une erreur est renvoyée. Si les mots de passe correspondent, l'utilisateur est authentifié.

8. JEU D'ESSAI

A. Test unitaire

Dans mon projet j'ai utilisé le framework Jest et la bibliothèque supertest. Ce test vérifie le bon fonctionnement de l'API /api/animations. Il s'assure que l'application peut se connecter à la base de données, que le serveur démarre correctement avant les tests et qu'il se ferme correctement après. Le test vérifie également que l'endpoint /api/animations renvoie bien un statut HTTP 200 et que les données sont sous la forme attendue (un tableau).

Supertest effectue des requêtes HTTP sur l'application, et Jest pour les assertions (c'est-à-dire vérifier si la réponse de l'API est celle attendue). Voici une explication détaillée du test unitaire concernant animationController (Cf Annexe 14):

- `describe` : C'est une fonction fournie par Jest pour regrouper des tests logiquement. Ici, tous les tests de l'Animation Controller sont groupés sous ce bloc.
- `beforeAll` : Cette fonction est exécutée avant tous les tests dans le bloc describe. Cela est utile pour effectuer des actions qui doivent être faites une seule fois avant que les tests ne commencent.
- `server = app.listen(5173)` : Cela démarre le serveur de l'application sur le port 5173 avant d'exécuter les tests.
- `await sequelize.authenticate()` : Cela vérifie que la connexion à la BDD est établie avec succès avant que les tests ne commencent.
- `afterAll` : Cette fonction est exécutée après tous les tests dans le bloc describe. Elle est utilisée pour nettoyer ou fermer la connexion à la BDD ou le serveur. Ici, `await sequelize.close()` : ferme la connexion à la BDD.
- `it` : Cette fonction est utilisée pour définir un test spécifique. Le premier argument est une description de ce que fait ce test. Le second argument est une fonction asynchrone qui contient le code du test lui-même. Ici, `request(app).get("/api/animations")` effectue une requête HTTP GET vers l'endpoint /api/animations de l'application (qui devrait renvoyer une liste d'animations). `expect(response.status).toBe(200)` vérifie que la requête a réussi. `expect(Array.isArray(response.body.data)).toBe(true)` vérifie que la réponse contient une propriété data qui est un tableau (Array).



Ensuite, on effectue la commande `pnpm run test` pour vérifier que tout fonctionne correctement.

B. Mettre à jour un profil (Cf Annexe 15)

Lorsque l'utilisateur modifie son profil, une série d'actions se déclenchent :

- Les données saisies dans le formulaire sur le front sont envoyées au back.
- Le back traite ces données et met à jour la BDD.
- Une fois que la BDD est mise à jour, le back envoie une réponse au front pour indiquer si l'opération a réussi.

- Côté frontend

Lorsqu'un utilisateur clique sur "Modifier" et soumet le formulaire, la fonction `handleProfileUpdate` est appelée. Cette fonction envoie une requête HTTP `PUT` avec les données du profil vers l'API, spécifiquement à l'URL : `router.put("/:userId", cw(profileController.updateProfile))`; puis les données envoyées sont au format JSON

- Côté backend

Le contrôleur `updateProfile` récupère l'ID de l'utilisateur depuis les paramètres d'URL (`req.params.userId`) et les données du profil depuis le corps de la requête (`req.body`).

On peut également vérifier dans l'api grâce à thunder client si la requête fonctionne.

- Côté BDD

La commande SQL utilisée dans le backend met à jour les données d'un utilisateur dans la table `profiles`. (En annexe un exemple de requête si je le faisais directement dans la BDD : La requête UPDATE agit comme suit :

- Elle recherche la ligne correspondant à l'id.
- Elle met à jour les colonnes avec les nouvelles valeurs fournies.
- Avec `RETURNING *`, elle renvoie la ligne mise à jour pour confirmer le succès de l'opération.

9. VEILLE SÉCURITÉ

A. Veille technologique

Si je souhaite que mon application évolue , je peux tester de nouvelles fonctionnalités et/ou améliorer ce qui est déjà en place. Par exemple, pour gérer des tokens plus complexes ou nécessitant des validations supplémentaires, je pourrai envisager d'intégrer une bibliothèque comme *jwt-decode* pour simplifier les opérations de parsing du token.

Lors de la réalisation de ce projet, j'ai réalisé une veille sur les vulnérabilités de sécurité liées à la gestion des tokens, en particulier dans les API et les applications web. Les tokens, comme les JSON Web Tokens (JWT), jouent un rôle essentiel dans l'authentification et l'autorisation des utilisateurs. Une mauvaise gestion des tokens peut introduire des vulnérabilités critiques, compromettant la confidentialité et la sécurité des données des utilisateurs. Lors de cette veille j'ai constaté un dysfonctionnement que je vais présenter ci-dessous.

B. Résolution de problème

Dans le cadre de ce projet, je me suis retrouvée confrontée à des problématiques, en l'occurrence l'expiration du token qui ne se faisait pas. J'ai dû m'aider de sources pour comprendre mon erreur, telles que https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Session_Management_Cheat_Sheet.md également, je me suis beaucoup appuyé sur les cours reçus avec O'Clock et leurs sources sur Github tout au long du projet. Pour l'exemple qui va suivre sur la récupération du token :

<https://github.com/O-clock-Pavlova/S16-orecipes-front-SoleneOclock/commit/8aae8ecc087b8fbce80c3eb41ed6f7509f3334aa>). Et j'ai aussi utilisé des outils d'intelligence artificielle comme ChatGPT pour m'aider à clarifier certaines questions techniques, par exemple sur les tokens JWT, la sécurité des API, et pour obtenir des conseils sur l'implémentation de certaines fonctionnalités. Ce recours à l'outil m'a permis de compléter mes recherches et d'enrichir mes choix de conception.

- Expiration insuffisante des tokens :



Ici, j'avais bien implémenté le token dans le back avec une durée d'expiration de 1h. Malgré ça, la déconnexion ne se faisait jamais. Je me suis rendue compte que dans le front le code ne vérifiait pas si le token avait expiré avant d'autoriser l'utilisateur à rester connecté. Si un token expiré est présent dans localStorage, l'utilisateur restera connecté tant que le front n'en valide pas l'état.

Pour garantir que l'utilisateur se déconnecte automatiquement après une heure d'inactivité, plusieurs modifications ont été apportées dans les fichiers *authService.ts* et *authContext.ts*.

- Modification dans *authContext.ts* :

L'objectif principal est de gérer l'expiration du token et de déconnecter automatiquement l'utilisateur si le token expire. J'ai ajouté la fonction *isTokenExpired* dans *authContext.ts* :

J'ai choisi d'utiliser la méthode *jwtDecode*, qui me paraît être une gestion plus simple.

Ici, le type de *decoded* est explicitement défini comme un objet contenant une clé *exp* de type number. Si la date d'expiration (*decoded.exp*) est strictement inférieure à *currentTime*, le token est expiré.

```
// Vérifie si le token est expiré
function isTokenExpired(token: string): boolean {
  try {
    const decoded: { exp: number } = jwtDecode(token);
    const currentTime = Math.floor(Date.now() / 1000);
    return decoded.exp < currentTime;
  } catch (error) {
    console.error("Erreur lors de la vérification du token:", error);
    return true;
  }
}
```

```

// Fonction de connexion
const loginUser = async (email: string, password: string) => {
  try {
    const response = await loginUserService(email, password);
    if (response.success && response.data) {
      const token = response.data.token;

      if (!isTokenExpired(token)) {
        setUser(response.data.user); // Mettre à jour l'utilisateur
        localStorage.setItem("token", token); // Stocker le token
        localStorage.setItem("user", JSON.stringify(response.data.user)); // Stocker les données
                                                utilisateur
      } else {
        throw new Error("Token expiré");
      }
    } else {
      throw new Error(response.message || "Échec de la connexion");
    }
  } catch (error) {
    console.error("Erreur lors de la connexion:", error);
    throw error;
  }
};

```

La fonction `loginUserService` envoie une requête au back pour obtenir un token après une connexion réussie. Le token reçu à la fonction `isTokenExpired(token)`. Cette fonction vérifie si le token est expiré en comparant la valeur d'expiration contenue dans le token (`decoded.exp`) avec l'heure actuelle. Si le token est expiré, la fonction renvoie true et ça déclenche une erreur. Si le token est valide et donc non expiré, on met à jour l'état de l'utilisateur (`setUser`), et on stocke le token et les informations utilisateur dans le localStorage pour qu'ils soient persistants entre les rechargements de la page.

CONLUSION

Le projet Survival-Parc a été une aventure enrichissante, tant sur le plan technique que sur le plan de la gestion d'un projet en autonomie. Ce projet m'a permis de mettre en pratique une large palette de compétences en développement, en gestion de base de données, et en sécurité des applications web apprises durant ma formation chez O'Clock.

Cependant, le projet Survival-Parc n'en est qu'à ses débuts, et de nombreuses évolutions sont envisageables pour enrichir l'expérience utilisateur et répondre à de nouveaux besoins. Par exemple, la base de données pourrait être améliorée pour intégrer des fonctionnalités avancées telles que :

- Ajout d'une table des tarifs permettant de proposer des prix différenciés en fonction de critères tels que la saison, les événements spéciaux ou les promotions.
- Spectacles et événements : Intégration d'une fonctionnalité pour afficher les spectacles et événements spéciaux. Une nouvelle table pourrait être ajoutée pour gérer ces événements.
- Système de paiement fictif : L'ajout d'un système de paiement permettrait de rendre l'application plus réaliste.

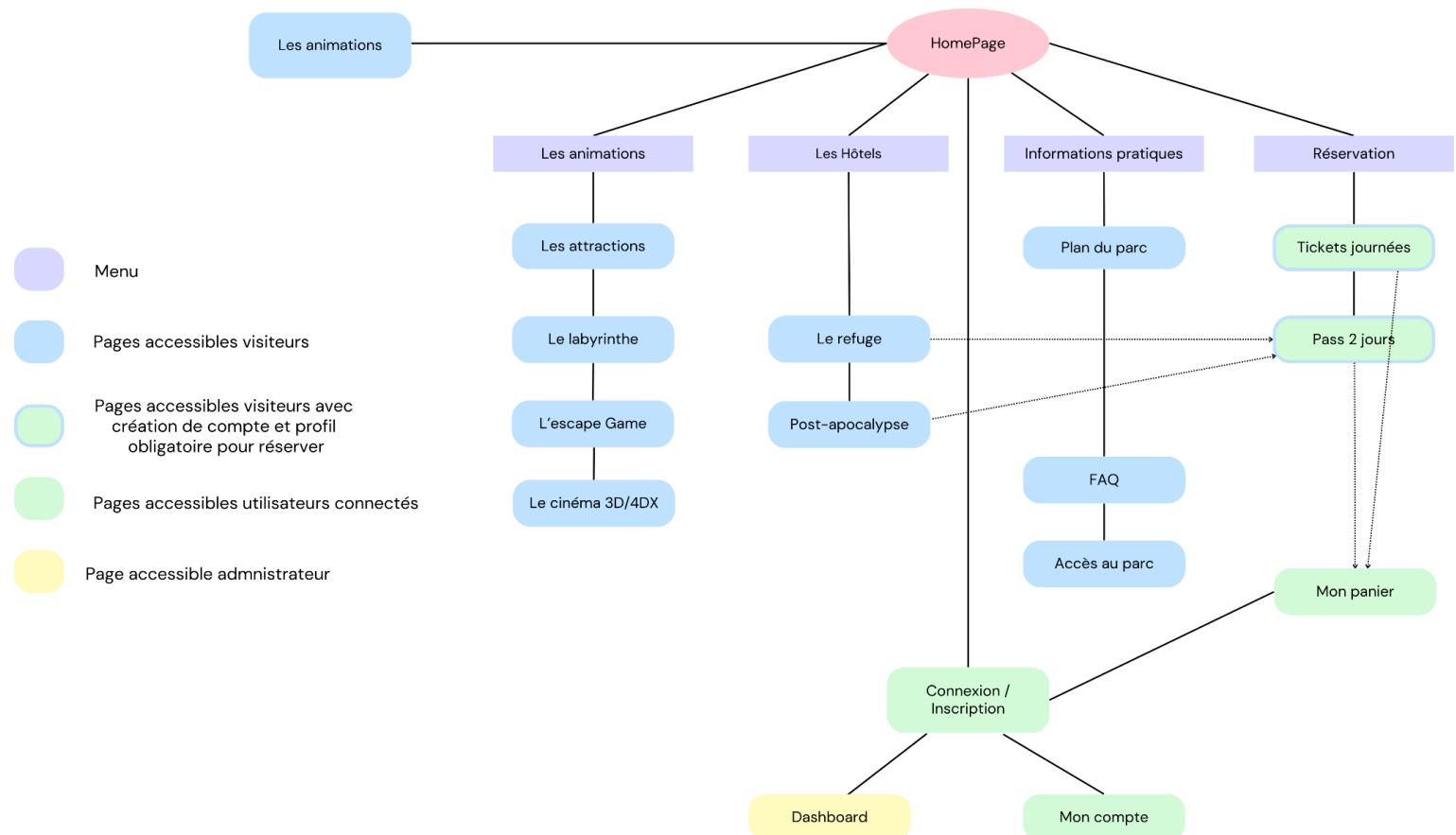
Ce projet a permis de construire une application fonctionnelle et sécurisée, tout en offrant de nombreuses pistes d'améliorations et d'enrichissements pour l'avenir. J'ai pu mettre en valeur mes compétences techniques et acquérir de nouvelles connaissances essentielles dans la gestion de projets web, en particulier en matière de sécurité et d'architecture de bases de données.

Cette formation très intensive a confirmé mon choix de réorientation professionnelle. Je souhaite à l'avenir continuer d'approfondir mes connaissances et compétences via des formations et de l'apprentissage en autonomie.

Par la suite, je souhaite créer et alimenter mon portfolio de projets bénévoles. J'aimerai approfondir Survival-parc, je dois créer un site de photos animalières et un site proposant les services d'un coaching sportif. En parallèle, je suis inscrite à France Travail et j'ai pour objectif de trouver un emploi en tant que développeuse Back.

ANNEXES

Annexe 1 : WORKFLOW





Annexe 2 : Userstories

En tant que	Je veux	Afin de
Visiteur non inscrit	parcourir les animations	voir les informations sur l'animation
Visiteur non inscrit	créer un compte	réserver / accéder à mon profil
Visiteur non inscrit	pouvoir effectuer une recherche	trouver des informations précises
Visiteur non inscrit	pouvoir contacter le support client	poser des questions
Visiteur non inscrit	accéder aux informations pratiques	connaître les horaires et l'accès au parc
Utilisateur inscrit	me connecter	Accéder à mon profil
Utilisateur inscrit	modifier mon profil	mettre à jour mes données personnelles
Utilisateur inscrit	supprimer mon compte	supprimer mes informations personnelles
Utilisateur inscrit	me connecter	Réserver des tickets/pass
Utilisateur inscrit	accéder aux informations pratiques	connaître les horaires et l'accès au parc
Utilisateur inscrit	pouvoir contacter le support client	poser des questions
Utilisateur inscrit	pouvoir effectuer une recherche	trouver des informations précises
Utilisateur inscrit	parcourir les animations	voir les informations sur l'animation
Utilisateur inscrit	pouvoir annuler en ligne ma réservation	éviter des réservations inutiles
Utilisateur inscrit	avoir le détail des mes réservations	garder une traçabilité
Administrateur	me connecter au dashboard	gérer le contenu du site
Administrateur	ajouter, modifier ou supprimer une animation	Maintenir le site à jour
Administrateur	voir la liste des réservations	Avoir une vue d'ensemble de l'activité du parc
Administrateur	gérer les comptes utilisateurs	assurer la sécurité et la maintenance du site
Administrateur	Modifier les prix	Adapter l'offre et la demande

Annexe 3 : Routes api

URL	Méthode HTTP	Controller	Méthode	Commentaire
api/auth/login	post	authController	findOne	se connecter
api/auth/logout	post	authController	findByPk	se déconnecter
api/user/register	post	userController	create	créer un compte
api/user	get	userController	findAll (where: { role: "admin" })	récupérer les users/admins
api/user/:id	get	userController	findByPk	récupérer un user
api/user/:id	put	userController	findByPk	modifier un user
api/user/:id	delete	userController	findByPk	supprimer un user
api/profile	get	profileController	findAll	récupérer les profils
api/profile/:userId	post	profileController	create	créer un profil
api/profile/:userId	get	profileController	findByPk	récupérer un profil
api/profile/:userId	put	profileController	findByPk	modifier un profil
api/profile/:userId	delete	profileController	findByPk	supprimer un profil



Annexe 3 : Routes api - suite

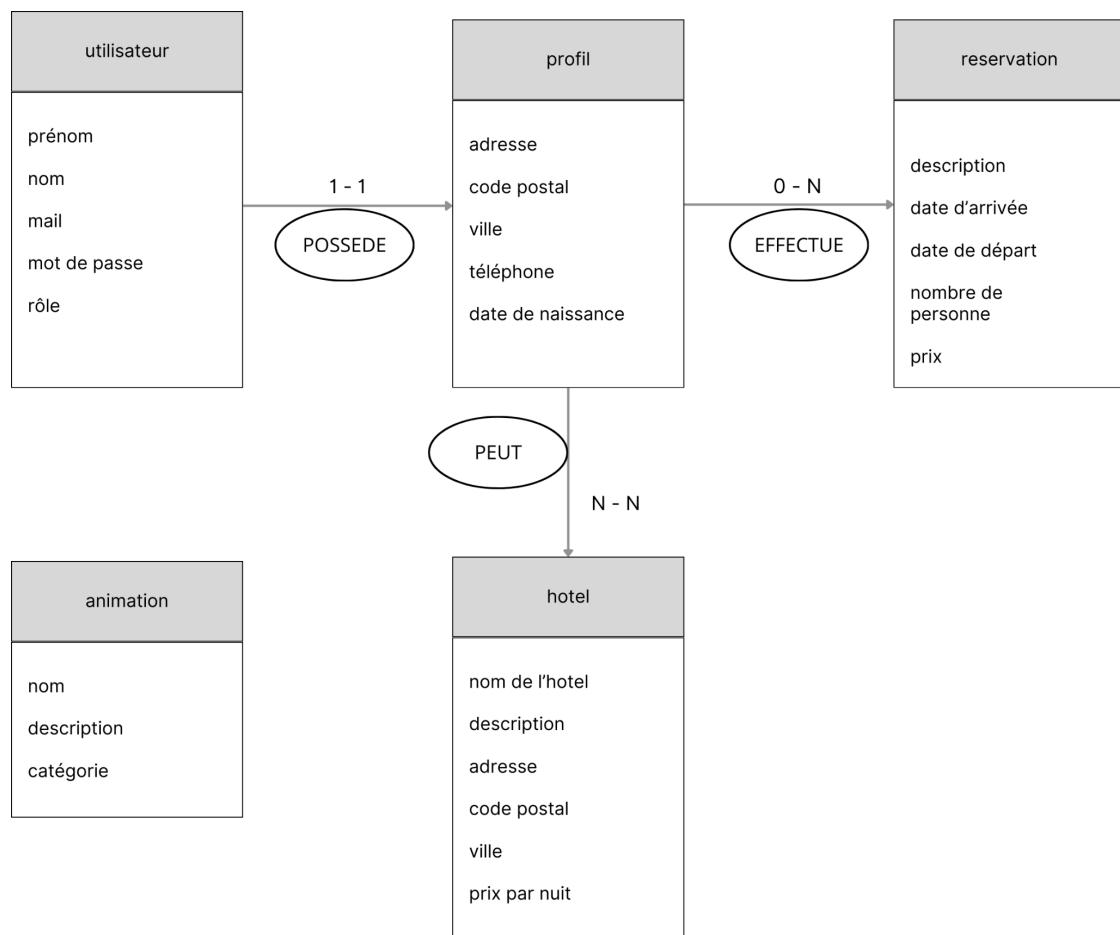
api/reservation	get	reservationController	findAll	récupérer les réservations
api/reservation/:userId	get	reservationController	findAll ({ where: { userId } });	récupérer les réservations d'un user
api/reservation/:userId	post	reservationController	create	créer une réservation
api/reservation/:userId	put	reservationController	findOne	modifier une réservation
api/reservation/:userId	delete	reservationController	findOne	supprimer une réservation
api/animations	get	animationController	findAll	récupérer les animations
api/animations/:type	get	animationController	findAll ({ where: { type } });	récupérer les animations par type
api/animations/:id	get	animationController	findById	récupérer une animation
api/animations	post	animationController	create	créer une animation
api/animations/:id	put	animationController	findById	modifier une animation
api/animations/:id	delete	animationController	findById	supprimer une animation

Annexe 3 : route API - suite

api/hotel	get	hotelController	findAll	récupérer les hôtels
api/hotel/:id	get	hotelController	findByPk	récupérer un hôtel
api/hotel	post	hotelController	create	créer un hôtel
api/hotel/:id	put	hotelController	findByPk	modifier un hôtel
api/hotel/:id	delete	hotelController	findByPk	supprimer un hôtel
api/profilehotel	get	profileHotelController	findAll	Récupérer toutes les réservations d'hôtel
api/profilehotel/:profileId	get	profileHotelController	findAll	Récupérer toutes les réservations d'hôtel d'un profil
api/profilehotel/:profileId	post	profileHotelController	findOne	Créer une réservation d'hôtel
api/profilehotel/:profileId	put	profileHotelController	findOne	Modifier une réservation d'hôtel
api/profilehotel/:profileId	delete	profileHotelController	findOne	Supprimer une réservation d'hôtel

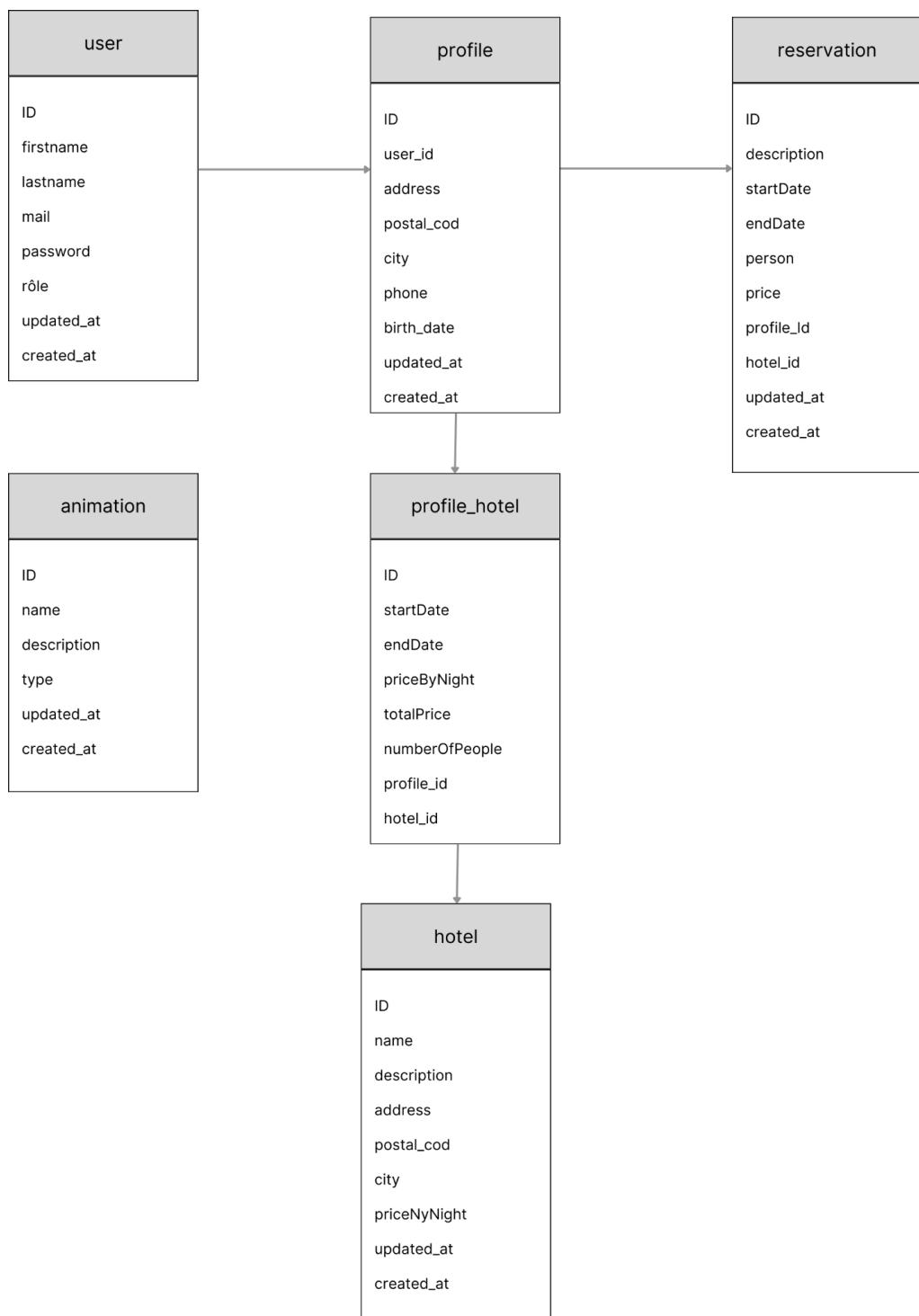


Annexe 4 : MCD





Annexe 5 : MLD



Annexe 6 : Dictionnaire des données

Table	Attribut	Type	Description
User	id	INT	Identifiant unique de l'utilisateur
User	firstName	VARCHAR(255)	Prénom de l'utilisateur
User	lastName	VARCHAR(255)	Nom de l'utilisateur
User	email	VARCHAR(255)	Email de l'utilisateur (unique)
User	password	VARCHAR(255)	Mot de passe de l'utilisateur
User	role	ENUM('admin','user')	
User	createdAt	TIMESTAMP	Date de création
User	updatedAt	TIMESTAMP	Date de mise à jour
Profile	id	INT	Identifiant unique du profil
Profile	userId	INT	Référence à l'utilisateur (clé étrangère)
Profile	firstName	VARCHAR(255)	Prénom du profil
Profile	lastName	VARCHAR(255)	Nom du profil
Profile	birthDate	DATE	Date de naissance du profil
Profile	phone	VARCHAR(20)	Numéro de téléphone
Profile	address	VARCHAR(255)	Adresse du profil
Profile	postalCode	VARCHAR(10)	Code postal de l'adresse
Profile	city	VARCHAR(255)	Ville de l'adresse
Profile	createdAt	TIMESTAMP	Date de création
Profile	updatedAt	TIMESTAMP	Date de mise à jour



Hotel	id	INT	Identifiant unique de l'hôtel
Hotel	name	VARCHAR(255)	Nom de l'hôtel
Hotel	description	TEXT	Description de l'hôtel
Hotel	address	VARCHAR(255)	Adresse de l'hôtel
Hotel	postalCode	VARCHAR(10)	Code postal de l'hôtel
Hotel	city	VARCHAR(255)	Ville de l'hôtel
Hotel	priceByNight	DECIMAL(10, 2)	
Hotel	createdAt	TIMESTAMP	Date de création
Hotel	updatedAt	TIMESTAMP	Date de mise à jour
Animation	id	INT	Identifiant unique de l'animation
Animation	name	VARCHAR(255)	Nom de l'animation
Animation	description	TEXT	Description de l'animation
Animation	type	VARCHAR(255)	Type d'animation (ex : sport)
Animation	createdAt	TIMESTAMP	Date de création
Animation	updatedAt	TIMESTAMP	Date de mise à jour

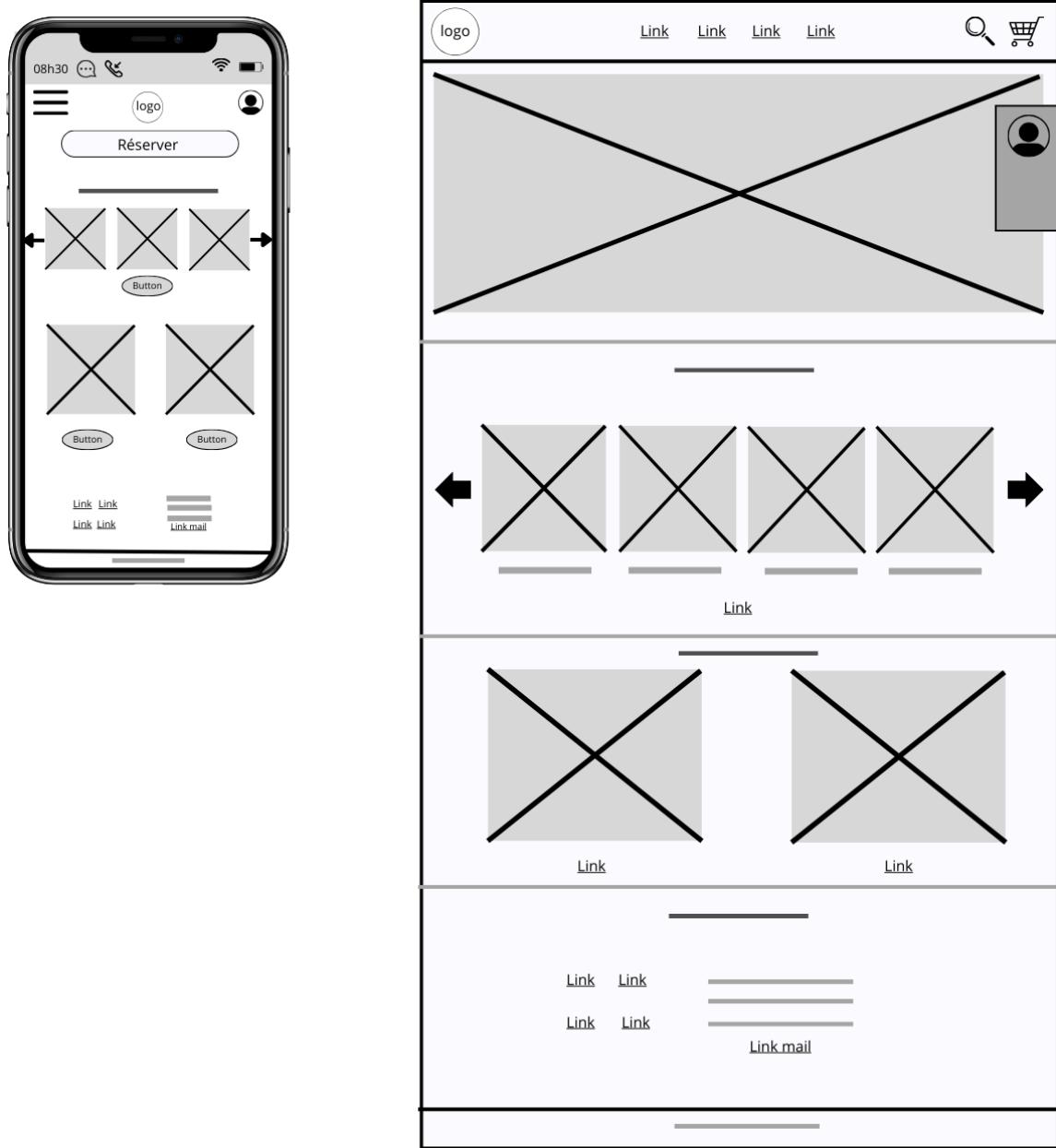
Reservation	id	INT	Identifiant unique de la réservation
Reservation	description	TEXT	Description de la réservation
Reservation	startDate	DATE	Date de début de la réservation
Reservation	endDate	DATE	Date de fin de la réservation
Reservation	person	INT	Nombre de personnes
Reservation	price	DECIMAL(10)	Prix total de la réservation
Reservation	profileId	INT	Référence au profil de l'utilisateur (clé étrangère)
Reservation	hotelId	INT	Référence à l'hôtel réservé (clé étrangère)
Reservation	hotelName	VARCHAR(255)	Nom de l'hôtel réservé (optionnel)
Reservation	createdAt	TIMESTAMP	Date de création
Reservation	updatedAt	TIMESTAMP	Date de mise à jour



profile_hotels	id	INTEGER	- Clé primaire - Auto-incrémentation (AUTO_INCREMENT) - Non nul (NOT NULL) - Identifiant unique de la réservation.
profile_hotels	profileId	INTEGER	- Clé étrangère vers profiles(id) - Non nul (NOT NULL) - Suppression en cascade (ON DELETE CASCADE).
profile_hotels	hotelId	INTEGER	- Clé étrangère vers hotels(id) - Non nul (NOT NULL) - Suppression en cascade (ON DELETE CASCADE).
profile_hotels	startDate	DATE	- Non nul (NOT NULL) - Date de début de la réservation.
profile_hotels	endDate	DATE	Non nul (NOT NULL) - Date de fin de la réservation.
profile_hotels	priceByNight	DECIMAL(10, 2)	- Non nul (NOT NULL) - Prix par nuit pour la réservation, avec 2 décimales
profile_hotels	numberOfPeople	INTEGER	- Non nul (NOT NULL) - Nombre de personnes incluses dans la réservation.
profile_hotels	totalPrice	DECIMAL(10, 2)	- Non nul (NOT NULL) - Prix total de la réservation
profile_hotels	status	STRING	- Non nul (NOT NULL) - Valeur par défaut : "pending" - Statut de la réservation (pending, confirmed, cancelled).
profile_hotels	createdAt	TIMESTAMP	Date de création
profile_hotels	updatedAt	TIMESTAMP	Date de mise à jour



Annexe 7 : Wireframe HomePage





Annexe 8 : Maquette HomePage

Homepage

The maquette displays the Survival PARC homepage layout across two devices: a smartphone on the left and a desktop browser on the right.

Annotations:

- Couleur de fond de la homePage : dégradé de black à #1F2937** (Color of the homepage background: black gradient to #1F2937) - Points to the top bar of the mobile phone and the top header area of the desktop page.
- Couleur de fond Header : black** (Color of the header background: black) - Points to the top header bar of the desktop page.
- Film d'animation** (Animation film) - Points to the large central image on the desktop page.
- Couleur de fond Cards : #374151** (Color of the cards background: #374151) - Points to the card area under "Nos Animations" on the desktop page.
- Couleur de fond footer : #111827** (Color of the footer background: #111827) - Points to the bottom footer bar of the desktop page.

Mobile Phone Layout (Left):

- Header: Survival PARC logo, Réserver button, navigation menu icon.
- Section: Nos Animations (with three thumbnail images and a "Découvrir les animations" button).
- Section: En savoir plus (with two thumbnail images).
- Footer: CGV, Mentions légales, Coordonnées (Adresse, Téléphone, Contact mail), Survival-parc tous droits réservés - 2024.

Desktop Browser Layout (Right):

- Header: Animations, Hotels, Infos pratiques, Réservations, search icon, shopping cart icon.
- Section: QURAAINTINE ZONE (Large banner image).
- Section: Nos Animations (with four thumbnail images and a "Découvrir les animations" button).
- Section: Les hôtels (with two thumbnail images and "En savoir plus" buttons).
- Footer: Coordonnées (CGV, Mentions légales), Survival-parc tous droits réservés - 2024.



Annexe 9 : extrait du code Responsive

```
/* Image en haut à gauche */


/* Image en haut à gauche */
    <div className="absolute top-16 sm:top-32 left-8 p-4 z-10">
        
    </div>

    /* Conteneur général */
    <div className="max-w-6xl w-full space-y-12 sm:space-y-16 mt-16 sm:mt-20 pt-8 sm:pt-12">
        /* Section titre */
        <motion.div
            className="text-center space-y-6 sm:space-y-8"
            initial={{ opacity: 0, y: -50 }}
            animate={{ opacity: 1, y: 0 }}
            transition={{ duration: 0.6 }}
        >
            <motion.h1
                className="text-3xl sm:text-5xl font-extrabold mb-4 text-gradient bg-clip-text bg-gradient-to-r from-green-400 to-yellow-500"
                initial={{ opacity: 0, y: -50 }}
                animate={{ opacity: 1, y: 0 }}
                transition={{ duration: 0.5 }}
            >
                {labyrintheAnimation[0].name}
            </motion.h1>
            <motion.p
                className="text-lg sm:text-xl italic text-gray-300 mb-12 sm:mb-20"
                style={{ marginTop: '60px', lineHeight: '1.8' }}
                initial={{ opacity: 0, y: -50 }}
                animate={{ opacity: 1, y: 0 }}
                transition={{ duration: 0.5, delay: 0.3 }}
            >
                {labyrintheAnimation[0].description}
            </motion.p>
        </motion.div>

        /* Section images */
        <motion.div
            className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-8 sm:gap-12"
            initial={{ opacity: 0, y: 50 }}
            animate={{ opacity: 1, y: 0 }}
            transition={{ duration: 0.6, delay: 0.7 }}
        >
            /* Image principale */
            <motion.div
                className="relative"
                initial={{ opacity: 0, scale: 0.8 }}
                animate={{ opacity: 1, scale: 1 }}
                transition={{ duration: 0.5 }}
            >
                
            </motion.div>

            /* Images supplémentaires */
            {[2, 3].map((index) => (
                <motion.div
                    key={index}
                    className="relative"
                    initial={{ opacity: 0, scale: 0.8 }}
                    animate={{ opacity: 1, scale: 1 }}
                    transition={{ duration: 0.5 }}
                >
                    <img
                        src={`${./src/assets/images/labyrinthe${index}.png`}
                        alt={`Image ${index}`}
                        className="w-full h-auto rounded-lg shadow-2xl transform hover:scale-105 transition-all duration-300"
                    />
                </motion.div>
            )))
        </motion.div>
    </div>


```



Annexe 9 Bis : Aperçu des pages responsives

Desktop :

The screenshot shows the desktop version of the Survival Parc website. At the top, there's a navigation bar with links for "Animations du parc", "Hôtels", "Infos pratiques", and "Réservations". On the far right of the header are "Connexion" and "Inscription" buttons. Below the header, a large banner features a green diagonal band with the text "ZONE INFECTÉE". The main title "Le passage secret" is centered above a descriptive paragraph: "Plongez dans un labyrinthe caché dans les ruines d'un hôpital psychiatrique abandonné. Ce lieu sinistre, envahi par la végétation, est un dédale de couloirs sombres et de murs recouverts de lierre. Au centre, une cour intérieure abrite un labyrinthe où l'obscurité et la brume jouent avec vos sens. Trouvez la sortie avant que la panique ne prenne le dessus." Below the text are three images showing different views of the abandoned hospital ruins and a large outdoor maze.

Version mobile 360x740

This screenshot shows the mobile version of the website at 360x740 dimensions. The layout is similar to the desktop version but adapted for smaller screens. The main title "Le passage secret" is centered above the same descriptive paragraph. The three images of the ruins and maze are displayed below the text. The top of the screen shows a status bar with "Dimensions: Galaxy S8" and other settings.

This screenshot shows the tablet version of the website at 1064x1366 dimensions. The layout is further condensed. The main title "Le passage secret" is centered above the descriptive paragraph. The three images are still present. The top of the screen shows a status bar with "Dimensions: iPad Pro" and other settings.

Version tablette 1064x1366



Annexe 10 : Model

```
import { Model, DataTypes } from "sequelize";

export class Hotel extends Model {
  static init(sequelize) {
    return super.init(
      {
        id: {
          type: DataTypes.INTEGER,
          primaryKey: true,
          autoIncrement: true,
          allowNull: false,
        },
        name: {
          type: DataTypes.STRING,
          allowNull: false,
        },
        description: {
          type: DataTypes.TEXT,
          allowNull: false,
        },
        address: {
          type: DataTypes.STRING,
          allowNull: false,
        },
        postalCode: {
          type: DataTypes.STRING,
          allowNull: false,
          validate: {
            len: [5, 10],
          },
        },
        city: {
          type: DataTypes.STRING,
          allowNull: false,
        },
        priceByNight: {
          type: DataTypes.DECIMAL(10, 2),
          allowNull: false,
        },
      },
      {
        sequelize,
        modelName: "hotel",
        tableName: "hotels",
        timestamps: true,
      }
    );
  }
  static associate(models) {
    // Un hôtel peut avoir plusieurs réservations
    this.belongsToMany(models.Profile, {
      through: models.ProfileHotel,
      foreignKey: "hotelId",
      as: "profiles",
    });
  }
}
```



Annexe 11 : Architecture

```
▽ BACK
  > docs
  > node_modules
  ▽ src
    ▽ config
      JS associations.js
      {} config.json
      JS dbclient.js
    > controllers
    ▽ middlewares
      JS authenticateToken.js
      JS error404.js
      JS errors.js
      JS success.js
      JS validate.js
    > models
    > routes
    > seeders
    ▽ utils
      JS cookieUtils.js
      JS ctrlWrapper.js
      JS jwt.js
    .env
    $ .env.example
    .gitattributes
    .gitignore
    {} .prettierrc
    O eslint.config.js
    JS index.js
    {} package-lock.json
    {} package.json
    ! pnpm-lock.yaml
    i README.MD
```



Annexe 12 : controller contenant la logique Login et Logout

```
● ● ●

const { User } = models;

// login
export const login = ctrlWrapper(async (req, res) => {
  const { email, password } = req.body;

  // Vérifie si l'email et le mot de passe sont fournis
  if (!email || !password) {
    return badRequestResponse(res, "Email and password are required");
  }

  // Recherche l'utilisateur par email
  const user = await User.findOne({ where: { email } });
  if (!user) {
    return error404(res, "User not found");
  }

  // Vérifie le mot de passe
  const isPasswordValid = await bcrypt.compare(password, user.password);
  if (!isPasswordValid) {
    return badRequestResponse(res, "Wrong password");
  }

  // Génère un token JWT pour l'utilisateur
  const token = generateToken(user); // Passer l'utilisateur complet

  // Sauvegarde le token dans la base de données pour l'utilisateur
  user.token = token;
  await user.save();

  // Définir le cookie avec le token
  setCookies(res, token);

  // Envoie une réponse avec succès
  successResponse(res, "Login successful", { user, token });
});

// logout
export const logout = ctrlWrapper(async (req, res) => {
  const { id } = req.body;

  // Recherche l'utilisateur par son ID
  const user = await User.findById(id);
  if (!user) return error404(res, "User not found");

  // Supprime le token de l'utilisateur pour déconnecter
  user.token = null;
  await user.save();

  // Effacer le cookie
  clearCookies(res);

  successResponse(res, "Logout successful");
});
```



Annexe 13 : Contrôle des champs avec Joi

```
● ● ●

import Joi from "joi";
import { badRequestResponse } from "../middlewares/errors.js";

export const validateLogin = (req, res, next) => {
  const schema = Joi.object({
    email: Joi.string().email().required().messages({
      "string.email": "format email invalide",
      "string.empty": "Email requis",
    }),
    password: Joi.string().min(6).required().messages({
      "string.min": "Le mot de passe doit comporter au moins 6 caractères",
      "string.empty": "Mot de passe requis",
    }),
  });

  const { error } = schema.validate(req.body);

  if (error) {
    return badRequestResponse(res, error.details[0].message);
  }

  next();
};
```



Annexe 14 : Test unitaire animationController

```
● ● ●

import request from "supertest";
import app from "../../index";
import { sequelize } from "../config/dbclient";

describe("Animation Controller", () => {
  let server;

  beforeAll(async () => {
    server = app.listen(5173); // Lancer le serveur avant les tests
    await sequelize.authenticate(); // Vérifier la connexion à la BDD
  });

  afterAll(async () => {
    await sequelize.close(); // Fermer la connexion à la BDD
    await new Promise((resolve, reject) => {
      server.close((err) => {
        if (err) {
          reject(err);
        } else {
          console.log('Server.closed');
          resolve();
        }
      });
    });
  }); // Attendre la fermeture du serveur
});

it("return 200 and all animations", async () => {
  const response = await request(app).get("/api/animations");
  expect(response.status).toBe(200);
  expect(Array.isArray(response.body.data)).toBe(true);
});
});
```



Annexe 15 : Mettre à jour un profil

Côté Front :

```
● ● ●

const handleProfileUpdate = async (updatedProfile: IProfile) => {
  try {
    const response = await axios.put(`http://localhost:3000/api/profile/${userId}`, updatedProfile);

    if (response.data.success) {
      setCreatedProfile(response.data.data); // Mettre à jour l'état avec les nouvelles données du
      profil
      setShowForm(false); // Fermer le formulaire
    } else {
      setError("Impossible de mettre à jour le profil.");
    }
  } catch (err) {
    console.error('Erreur lors de la mise à jour du profil:', err);
    setError("Erreur lors de la mise à jour des données.");
  }
};
```

Côté Back :

```
● ● ●

export const updateProfile = ctrlWrapper(async (req, res) => {
  const { firstName, lastName, email, phoneNumber, address } = req.body;
  const { userId } = req.params; // Utiliser userId dans l'URL

  const profile = await Profile.findOne({ where: { userId } }); // Trouver le profil par
  userId;
  if (!profile) {
    return error404(res, "Profile not found");
  }

  // Mettre à jour les champs si les valeurs sont fournies dans la requête
  if (firstName) profile.firstName = firstName;
  if (lastName) profile.lastName = lastName;
  if (email) profile.email = email;
  if (phoneNumber) profile.phoneNumber = phoneNumber;
  if (address) profile.address = address;

  // Sauvegarder les changements
  await profile.save();

  // Retourner une réponse de succès
  successResponse(res, "Profile updated successfully", profile);
});
```

Côté BDD :

```
● ● ●

UPDATE profiles
SET first_name = 'Jean', last_name = 'Dupont', birth_date = '1985-06-15',
phone = '0987654321', address = '456 Rue Modifiée', postal_code = '75002',
city = 'Paris'
WHERE id = 1
RETURNING *;
```