Pelops

Une application d'organisation d'activités sportives entre particuliers

Travail de Bachelor

Antoine Demont

Avril 2022

Supervisé par :

Prof. Dr. Jacques Pasquier-Rocha
and
Pascal Gremaud
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Table des matières

Liste des figures

Liste des codes source

1 Introduction

1.1. Motivation et Buts

Cette thèse de Bachelor présente la création d'une application d'organisation de rencontres sportives amateures. Elle s'inscrit dans le cadre d'un réel projet baptisé *Pelops* sensé voir le jour à l'été 2022. Le but de ce travail est de créer tous les composants nécessaires à la création de celle-ci, autant l'application en elle-même que le serveur devant gérer les requêtes des utilisateurs.

Ce serveur est une API REST et doit donc en respecter les caractéristiques.

1.2. Organisation

Introduction

L'introduction contient la motivation et les buts de ce travail, un bref récapitulatif de la structure de chaque chapitre ainsi qu'une description des conventions de formatage.

Chapitre 2 : Le projet Pelops

Ce chapitre présente le projet *Pelops*, de son origine à aujourd'hui. Une description du fonctionnement de l'application y est aussi incluse.

Chapitre 3 : Réalisation du projet

Ce chapitre décrit les moyens utilisés pour la réalisation du projet. Cela va des techniques d'organisation aux différentes technologies utilisées pour le développement de l'application.

Chapitre 4 : Implémentation

Ce chapitre explique plus en détails quelques points clés du code écrit pour créer l'API REST et l'application pour smartphone.

Chapitre 5 : Conclusion

Ce dernier chapitre résume les différents points traités dans ce rapport et donne quelques complications rencontrées durant le développement de ce projet ainsi que certaines directions dans lesquelles l'améliorer.

Appendices

Contient une liste d'acronymes communs, un lien vers le repository github du projet ainsi que les références du projet.

1.3. Notations and Conventions

— Conventions de formatage :

  — Les abréviations et acronymes sont notés Internet Protocol (IP) pour le permier usage et uniquement IP ensuite ;

  — Le code est formaté comme suit :

```
1 public double division(int _x, int _y) {
2     double result;
3     result = _x / _y;
4     return result;
5 }
```

— Une figure s ou un listings s sont numérotés dans un chapitre. Par exemple, un référence vers la figure j du chapitre i sera notée *fig. i.j.*

2
Le projet Pelops

2.1. Aperçu

Ce chapitre décrit le projet Pelops, de son origine à son statut actuel. Notamment la levée de fond réalisée afin de permettre à l'application d'un jour voir le jour sous une version commerciale et non pas un simple projet étudiant. Une description de l'application et son fonctionnement est aussi incluse à la fin de ce chapitre.

2.2. Origine du projet

Toute personne pratiquant un sport d'équipe s'est déjà retrouvée dans la situation où il manque un joueur pour équilibrer les équipes, ou alors à chercher un groupe dans une ville qu'elle ne connaît pas. C'est de ce constat qu'est né le projet Pelops. L'idée, créer une application permettant aux utilisateurs de créer leurs propres activités et de les partager avec les autres potentiels participants inscrits à Pelops.

2.3. Pelops aujourd'hui

En parallèle de la réalisation de ce travail, le projet Pelops a continué à se développer. En effet, un aspect important de ce genre d'applications est leur communauté. Plus la communauté d'utilisateurs est grande, plus elle sera amenée à grandir. C'est ce qu'on appelle un effet de réseau en termes économiques [2], où la valeur d'un bien, contrairement au cas classique, augmente à mesure que l'offre augmente. Dans le but d'établir sa

communauté le plus tôt possible, Pelops a d'abord été créé sur le réseau social *Instagram*. Avec plus de 2200 abonnés, de nombreux tournois et autres activités sportives ont étés organisés, rencontrant un succès ainsi qu'un intérêt populaire grandissant. Cet intérêt s'est notamment exprimé lors d'une montée de fonds qui a amené la somme de 25'600 francs en l'espace d'un mois.

2.3.1. Crowdfunding Pelops

Le *Crowdfunding*, ou financement participatif, est une méthode de financement à la mode parmi les start-ups et autres projets entrepreneuriaux. En effet, cela permet à une idée d'obtenir des fonds pour sa réalisation sans l'implication d'une banque ou la prise d'un crédit. Diminuant ainsi les risques pour l'entreprise menant cette campagne qui ne s'engage qu'avec des personnes croyant en son projet.

Pour Pelops, ce crowdfunding a été mené sur la platforme *Wemakeit*. Utiliser un site tiers pour la gestion de la levée de fonds apporte une crédibilité ainsi qu'une sécurité pour les investisseurs potentiels. De plus, concept répandu du crowdfunding, chaque personne choisissant de supporter le projet ne fait en fait qu'une promesse de versement. Cela signifie que le montant promis ne sera prélevé que si l'objectif défini par le créateur du projet est atteint. Cette gestion est garantie par la plateforme et évite ainsi les abus.

L'usage lors de crowdfunding est de définir différents paliers correspondants à un montant investi. Pour chaque palier, le leveur de fonds propose ainsi une récompense en cas de succès. Cela aide à engager les potentiels investisseurs et limite l'impression de simplement dépenser son argent dans quelque chose d'abstrait. Dans le cas de Pelops, ces récompenses allaient du T-shirt, visible à la figure 2.1, à l'intégration même dans l'application, ce qui peut s'apparenter à du sponsoring dans le sport ou l'art, ou alors à un abonnement permettant l'accès gratuit aux activités payantes organisées sur l'application [1].

[1] Liste complète des récompenses : `https://wemakeit.com/projects/pelops`

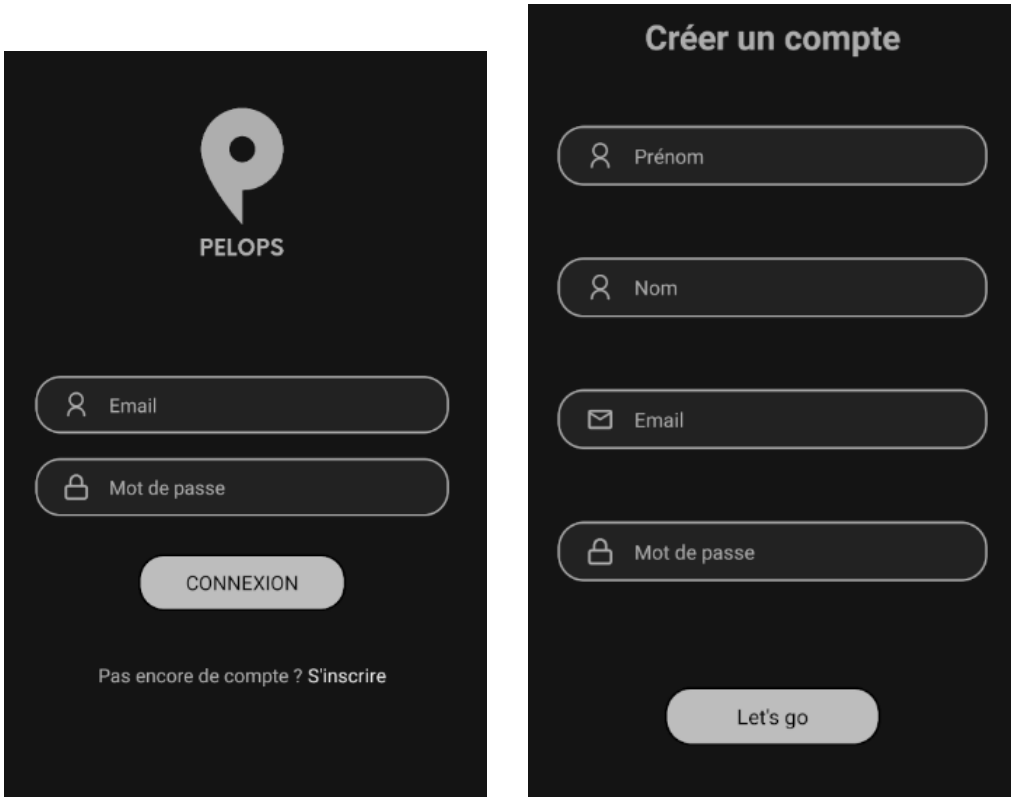
FIGURE 2.1. – T-shirt cadeau offert aux participants du crowdfunding

2.4. L'application

La base de l'application se compose d'une carte interactive où sont affichées les activités créées par les différents utilisateurs (fig. 2.4), d'où il est aussi possible de s'y inscrire ainsi que de créer une nouvelle activité. En parallèle à cette carte, il est aussi possible de consulter les activités qu'on a créées ou celles auxquelles l'on est inscrit (fig. 2.6). Depuis cet écran, il est possible pour l'utilisateur de se désinscrire d'une activité ou alors supprimer une activité dont il est le créateur. Les activités peuvent être triées soit alphabétiquement soit par date, ce qui permet à l'utilisateur de mieux les organiser. Par défaut, elles sont affichées par ordre d'inscription. Le dernier onglet, figure 2.3a, de l'application contient les informations de l'utilisateur ainsi que la possibilité de se déconnecter ou de supprimer son compte, ce qui aura pour conséquence de supprimer aussi toute activité dont il est l'organisateur. Il y trouvera aussi Dans la figure 2.8, on peut voir une description des interactions entre les différents modules qui composent l'application.

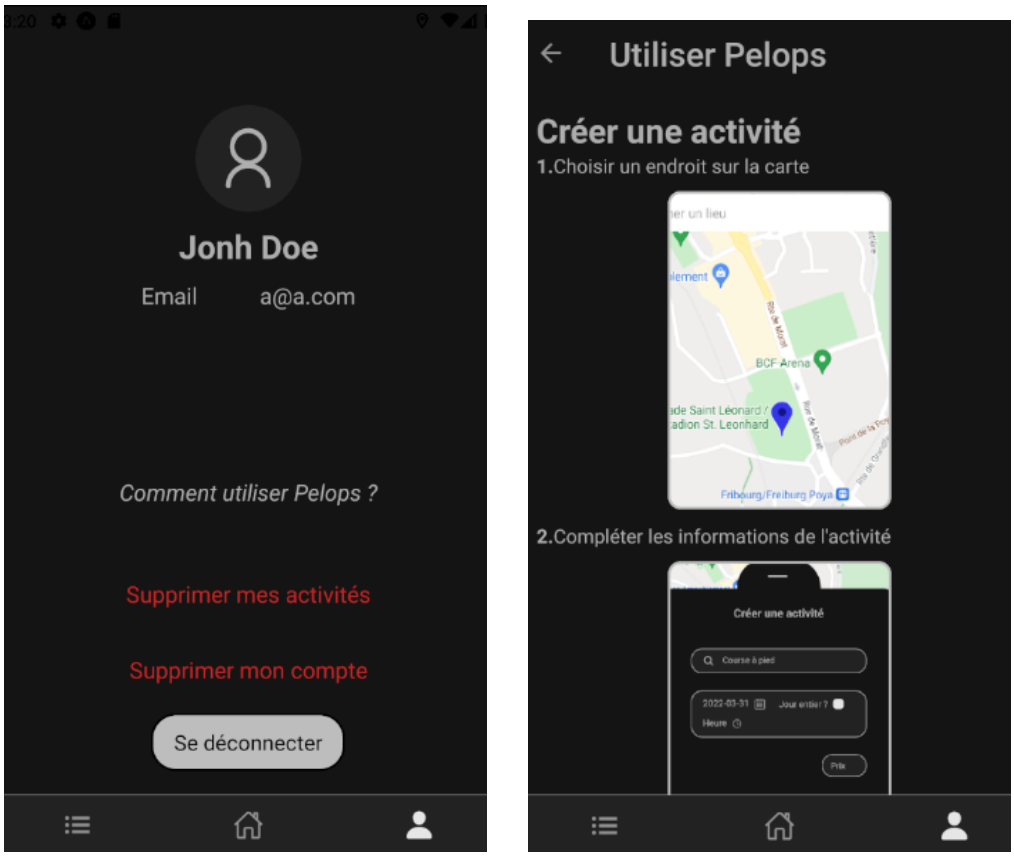
Une utilisation normale de l'application, et donc une traversée standard de ce graphe, serait en premier de se créer un compte et de se connecter (fig. 2.2).Une fois connecté, l'utilisateur peut consulter un tutoriel d'utilisation de l'application s'il n'est pas familier avec son fonctionnement (fig. 2.3). Ensuite, depuis la carte principale ou en cherchant un lieu depuis la barre de recherche (fig, 2.4), de créer (fig. 2.5) ou s'inscrire à des activités (fig. 2.7a). Il est ensuite possible de gérer ses activités (fig. 2.6), il est ici possible de

supprimer une activité, s'en désinscrire ou d'annuler l'inscription d'un participant (fig. 2.7 b) et c)).



(a) Écran d'accueil de l'application (b) Création d'un compte

FIGURE 2.2. – Écrans de démarrage de l'application



(a) Écran de gestion du compte (b) Écran d'explication

FIGURE 2.3. – Écrans d'information

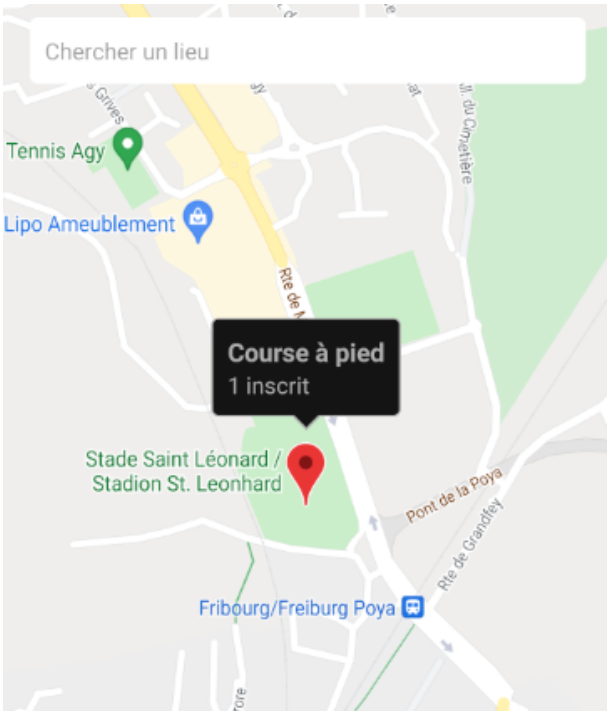


FIGURE 2.4. – Carte avec une activité créée par un autre utilisateur

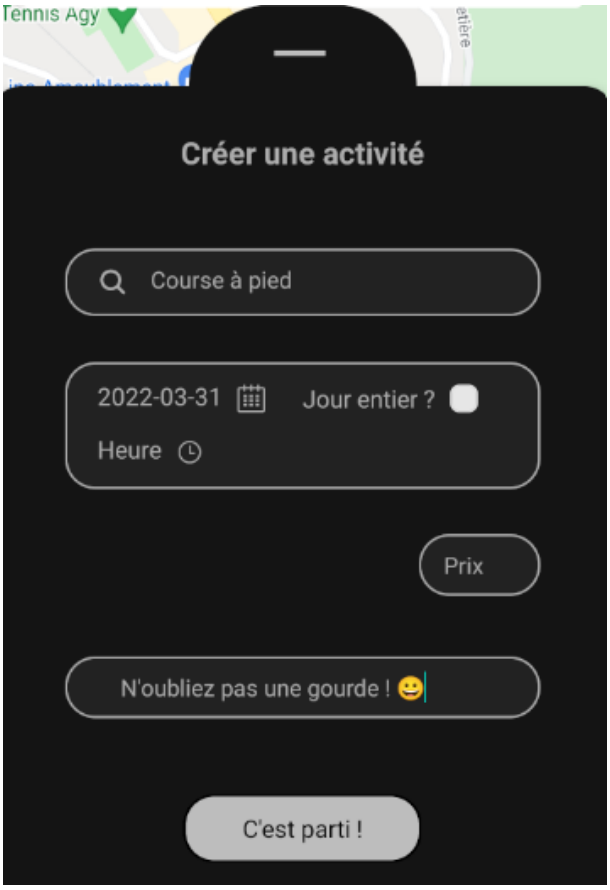
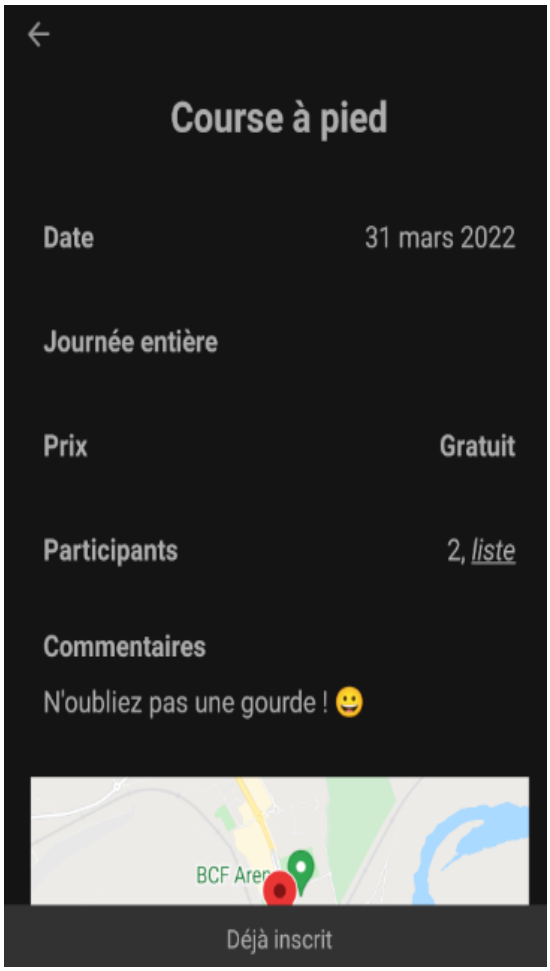
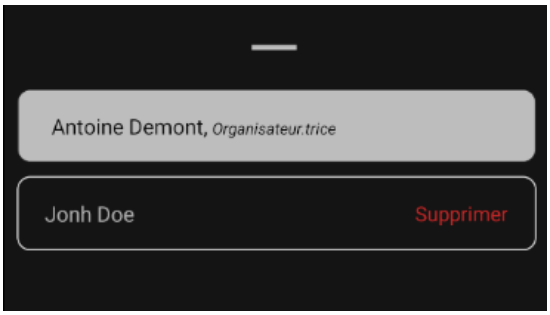


FIGURE 2.5. – Création d'une activité

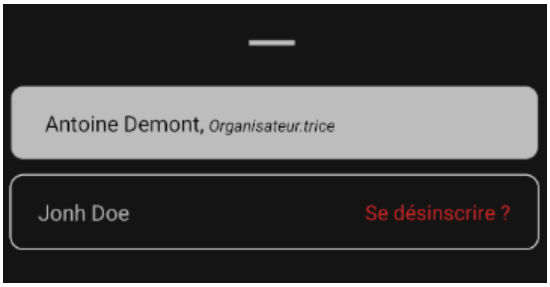
FIGURE 2.6. – Écran de gestion des activités



(a) Informations sur une activité



(b) Gérer la liste de participants en tant que créateur



(c) Gérer son inscription

FIGURE 2.7. – Détails d'une activité

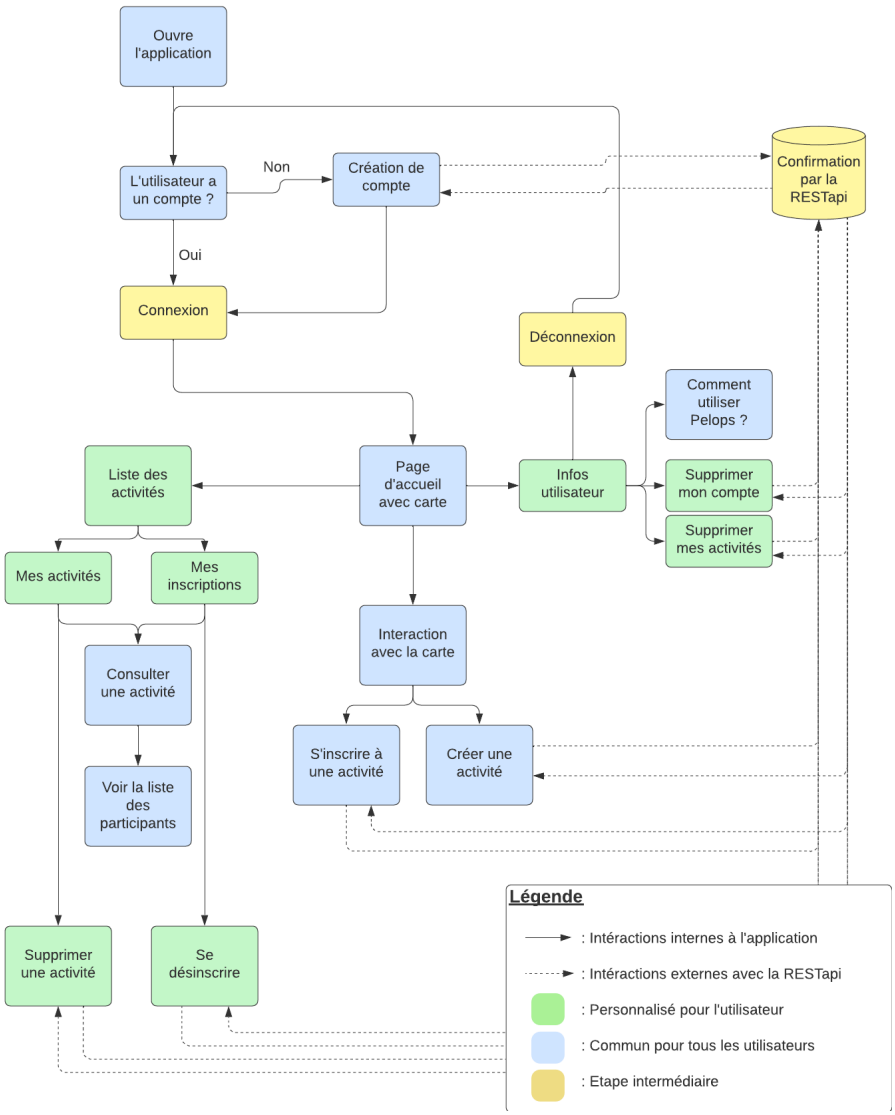


FIGURE 2.8. – Flowchart de l'application

2.4.1. Exemple d'utilisation de l'application

Afin de mieux illustrer l'application Pelops et son fonctionnement, un exemple concret s'impose. Prenons Alice, une étudiante de l'université qui vient d'arriver dans la ville de Fribourg, elle aime le basketball mais n'a pas le temps de jouer régulièrement dans un club. Alice connaît quelques personnes mais pas assez pour être assez de joueurs pour faire un match. Elle ouvre l'application Pelops et se crée un compte. Sur la carte, elle voit que des gens ont prévu d'aller jouer sur un terrain pas loin de chez elle. Alice profite de l'occasion pour s'y inscrire. Grâce à l'application elle peut aussi voir le nombre de personnes inscrites ainsi que leurs noms. Malheureusement elle doit se désinscrire à cause de son travail en tant que serveuse, mais elle a pu prendre contact avec les autres joueurs afin de s'organiser un match un autre jour.

Du côté du créateur d'une activité, prenons cette fois Bob, qui a pour habitude d'aller courir après sa journée de travail en tant que Banquier à Berne. Étant le plus jeune de

ses collègues, il fait sa course seul chaque jour. Se disant qu'avoir un groupe l'aiderait à se motiver d'avantage. Il se connecte donc sur l'application Pelops et crée une activité de course à pied à proximité de son lieu de travail à l'aide de la carte interactive. Grâce à Pelops, il a pu se trouver des partenaires motivés afin de préparer au mieux ses prochaines courses.

3
Réalisation du projet

3.1. Aperçu

Cette partie traite des aspects organisationnels et techniques du projet. Il y est décrit la méthodologie utilisée afin de rester productif et organisé ainsi que certaines étapes qui peuvent être préparées avant le début du développement comme l'utilisation de prototypes. Quant à la technique, cette partie présente le fonctionnement des différents outils utilisés pour la création de l'Application Programming Interface (API) Representational State Transfer (REST) ainsi que du client mobile.

3.2. Gérer un projet

Ce projet ayant eu lieu durant la crise du Coronavirus, les méthodes de travail devaient elles aussi être adaptées à cette situation particulière. En effet, sans un rythme de travail régulier et le remplacement de certains cours par des vidéos laissant libre l'étudiant de choisir ses heures d'étude, il peut devenir difficile de s'organiser et de définir des délais pour les tâches à réaliser.

Afin de clarifier la progression ainsi que donner un réel sentiment d'avancement, ce qui ne doit pas être négligé lorsque l'on travaille sur un projet de grande envergure, on peut décomposer composantes du projet en de multiples tâches de plus petites envergures. Ce procédé est similaire à la méthodologie *Scrum* [14], à la différence que dans ce cas

il s'agit d'organiser le travail d'une seule personne et non d'un groupe. Pour les non-initiés, la méthodologie *Scrum* consiste en une méthode d'organisation et de répartition de la charge de travail entre un groupe de développeurs et un client. La méthode est très complète mais un de ses aspects qui nous intéresse dans le cadre de ce projet est le concept de *sprints*. Au début du projet, les fonctionnalités attendues par le client sont formulées sous la forme d'histoires, par exemple, "Je veux pouvoir m'inscrire à une activité". Ces histoires sont ensuite découpées en tâches et reçoivent un montant de points indiquant l'estimation du temps requis pour les réaliser. Une fois que l'on a une liste de tâches, on peut définir différents *sprints* durant lesquels l'on va réaliser les étapes associées à un ensemble d'histoires durant une période de temps prédéfinie, habituellement de quelques semaines. Il existe de nombreux outils qui permettent d'encadrer cette méthodologie comme le site web *taiga* par exemple.

De cette méthodologie, j'ai décidé de reprendre le concept de découper un module en sous-tâches afin d'avoir une vue d'ensemble de mon avancement ainsi qu'une meilleure représentation du fonctionnement de chacun de ces composants. Étant seul sur ce projet, je me suis plutôt tourné vers le site web *Trello*, car les autres aspects de la méthodologie *Scrum* ne m'étaient pas indispensables.

3.2.1. Utiliser Trello

Trello est un site internet permettant de créer des cartes et de les grouper tout en ayant une interface simple, un exemple de son utilisation est visible à la figure 3.1.

Étant donné que mon projet se compose d'une application mobile ainsi que d'une API REST, j'ai défini un code couleur permettant d'identifier la partie du projet à laquelle chaque carte se reportait :

- **Rouge** : Application mobile
- **Bleu** : API REST
- **Vert** : Application mobile et API REST

Ensuite, chaque carte pouvait être associée à un groupe selon son statut. Au début, si l'on observe la figure 3.1, toutes les cartes se trouvent dans le premier groupe, appelé *Todo*, signifiant qu'elle contient une fonctionnalité qui doit être ajoutée au projet. Lorsqu'une tâche est commencée, sa carte est déplacée du premier groupe vers le second, *En cours*. Depuis ce groupe, une carte peut être terminée et envoyée vers le groupe *Terminé* ou mise en pause si sa fonctionnalité dépend d'autres pas encore implémentées, dans ce cas elle rejoindra le groupe *En pause*. Les derniers groupes, *Rédaction* et *Bugs*, contiennent respectivement les idées de sujets à traiter dans ce rapport et les différents bugs trouvés dans le code à résoudre.

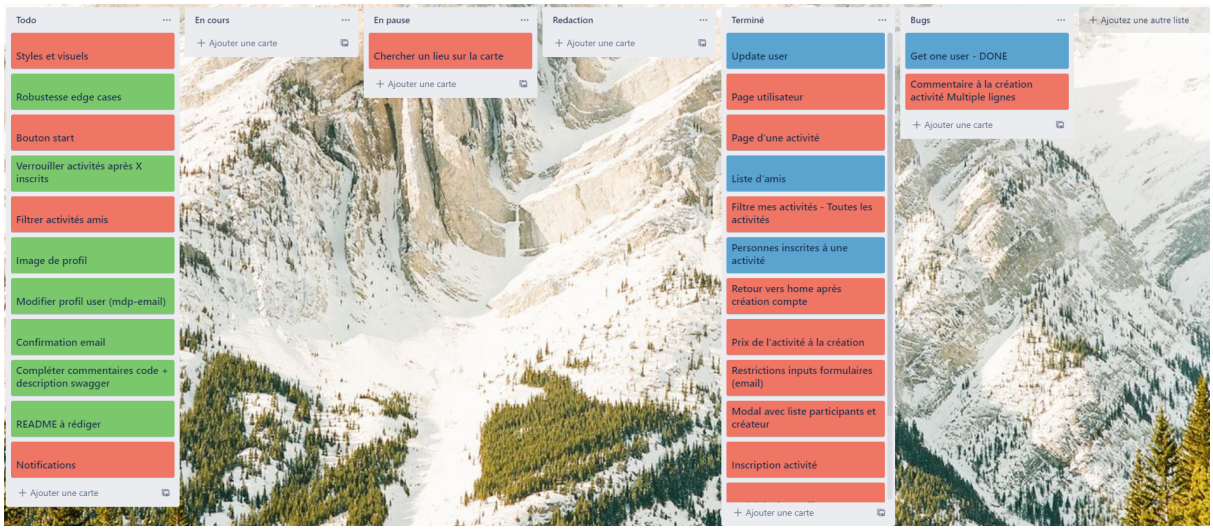


FIGURE 3.1. – Exemple d'utilisation de Trello

3.2.2. Un prototype de moyenne fidélité

Lors de la création d'un client web ou mobile, il est utile de commencer par un prototype de basse ou moyenne fidélité. Cela permet de réduire les coûts de développement lors de retour de clients. Idéalement, les utilisateurs doivent être inclus durant l'entier du processus de développement. On appelle cela *User Centered Design*. Concrètement, on fait tester un prototype à des utilisateurs en attendant leur retour avant de commencer le développement du système. En effet, la complexité réduite du prototype permet de plus simplement le modifier que s'il s'agit de changer le code ou l'architecture entière du système. Ces prototypes peuvent être de basse fidélité, comme par exemple une série de dessins illustrant grossièrement l'apparence du système ainsi que ses différentes interactions ou alors de moyenne et haute fidélité. Dans ces cas, l'utilisateur peut interagir avec le système comme s'il avait sous la main un produit fini. Leur degré de fidélité est défini par la complétion des fonctionnalités du prototype.

Dans le cadre de ce projet, on m'a fourni un prototype de moyenne fidélité réalisé avec AdobeXD [1] (fig. 3.2). La personne qui l'a réalisée est la même que celle responsable du projet Pelops présenté dans le chapitre 2. Ne faisant pas partie de mon travail je ne vais pas décrire plus en détail la création de ce prototype dans ce rapport.

Ce prototype permet au développeur de directement avoir une idée des composants qu'il doit ajouter dans son code ainsi que leur position. Grâce à cela, il peut se concentrer sur le fonctionnement sans avoir à faire de changement sur la disposition de ceux-ci à l'écran, gagnant ainsi un temps précieux. Il permet aussi d'estimer le travail restant. En effet, il peut être difficile de se représenter toutes les fonctionnalités du produit fini durant son implémentation.

[1] AdobeXD est un outil permettant le prototyping de l'interface de sites web, applications, jeux ou autres par la simple méthode du *drag and drop*. Plus d'informations sur leur site `https://www.adobe.com/ch_fr/products/xd.html`

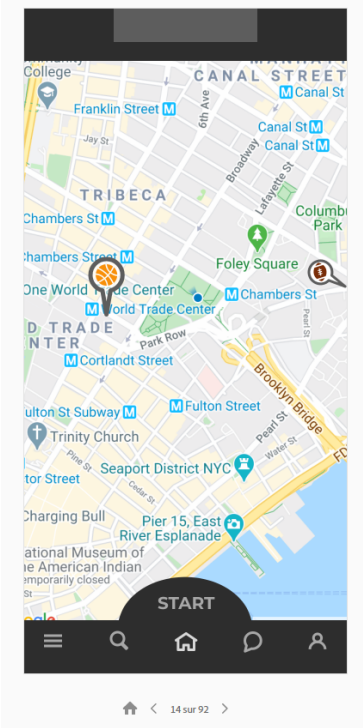


FIGURE 3.2. – Prototype moyenne fidélité de l'application

3.3. Technologies utilisées

Cette partie traite des différentes technologies utilisées dans ce projet. Il est composé de trois composants principaux, un *backend* composé d'une API REST ainsi que d'une base de données et le *frontend* lui contient le client mobile. Dans la figure 3.3, on peut aussi voir une partie responsable de l'authentification. Ce module est partiellement inclus dans l'API REST mais est assez différent dans son fonctionnement pour être représenté séparément de celle-ci dans la figure afin qu'il ne soit pas ignoré.

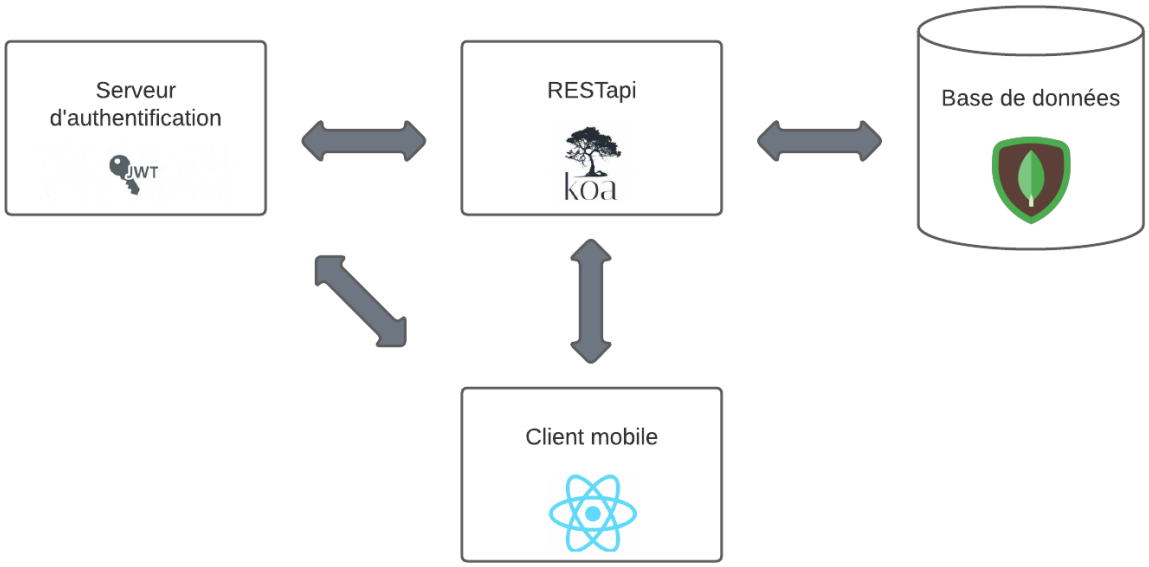


FIGURE 3.3. – Composants principaux du projet

3.3.1. Base de données

La base de données est responsable du stockage des informations des utilisateurs ainsi que des activités créées. Elle est construite avec *MongoDB* [6].

MongoDB

MongoDB [6] est un Database Management System (DBMS), c'est à dire un système de gestion de base de données. On peut séparer les DBMS selon le type de stockage des données, les plus commun sont les Relational Database Management System (RDBMS) qui stockent les données et les différentes relations entre elles mais il en existe d'autres comme les Times Series Database Management System (TSDBMS) qui stockent les données pour une durée déterminée, ou alors sur le langage utilisé dans le système, où l'on différencie principalement le *NOSQL* du Structured Query Language (SQL).

Selon ces critères, *MongoDB* est un *NOSQL* DBMS relationnel. En alternative à SQL, *MongoDB* utilise des documents JavaScript Object Notation (JSON) regroupés en collections selon les relations entre les données contenues dans ces documents (fig. 3.4).

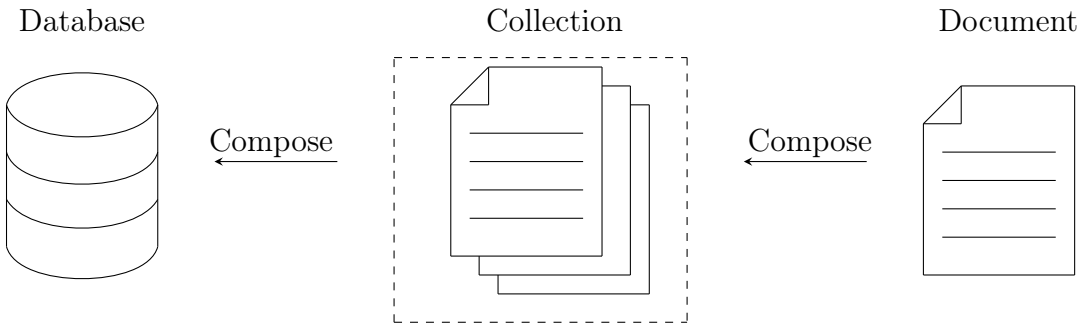


FIGURE 3.4. – Hiérarchie de MongoDB

Utiliser MongoDB

Chaque document contient un *modèle* représentant les données stockées dans la base de données. Dans le cadre de ce projet les deux modèles sont un pour les utilisateurs et un autre pour les activités. Ces modèles décrivent la structure que doivent prendre les données ajoutées à un document.

On voit par exemple dans le listing 3.1, que le sport est un *string* et est obligatoire pour être ajouté à la base de données. Au contraire, étant donné qu'une activité peut durer la journée entière, l'ajout d'une heure n'est pas obligatoire et donc ce champs peut être laissé vide lors de l'ajout de données. L'aspect relationnel de *MongoDB* peut être aperçu ligne 17 où il est fait une référence vers le modèle d'un utilisateur afin d'identifier le créateur de l'activité.

```
1        sport: {
2            type: String,
3            required: true,
4        },
5        date:{
6            day: {
7                type: Date,
8                required : true,
9            },
10           hour: {
11               type: String,
12           }
13       },
14       creator: {
15           type: Number,
16           required: true,
17           ref: 'User',
18       },
```

Listing 3.1 – Extrait du modèle d'une activité

3.3.2. API REST

L'API REST est la pièce centrale responsable des interactions entre les systèmes présentés dans la figure 3.3. C'est ici que sont gérées les requêtes vers la base de données ou l'ajout ainsi que la suppression de données. Certaines opérations sont sensibles et donc doivent

être protégées derrière un système d'authentification. C'est pourquoi la limite entre ce composant et le serveur d'authentification est floue comme mentionné précédemment. L'authentification et son fonctionnement sont traités dans la section du même nom.

Service REST

REST est une manière de représenter et transférer des documents, sous la forme de ressources. Cela repose sur plusieurs principes [12] [9] :

- Le système a une architecture client-serveur, communicant via Hypertext Transfer Protocol (HTTP).
- Seule une représentation des ressources est transférée, signifiant ainsi que celles-ci sont totalement découplées. Concrètement, le contenu d'une ressource peut être représenté au format eXtensible Markup Language (XML), JSON ou texte.
- La communication est sans état. Cela signifie que les requêtes d'un client ne sont pas stockées. Du *caching* est toutefois possible afin d'optimiser les interactions clients-serveur.

Chaque ressource est référencée par un Unified Resource Identifier (URI) de la forme de la figure 3.5. Elle peut ensuite être transmise via HTTP grâce aux méthodes standards Create, Read, Update, Delete (CRUD) [2]. Par exemple, dans le listing 3.2, on voit un exemple de requête HTTP qu'un client pourrait faire vers le serveur afin d'obtenir la liste des utilisateurs. Le *flag* "Accept" indique au serveur le format de fichier que veut obtenir en retour le client, ici un JSON.

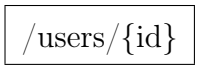


FIGURE 3.5. – URI d'un utilisateur

```
GET /users/ HTTP/1.1
Host: localhost
Accept: application/json
```

Listing 3.2 – Requête GET de la liste des utilisateurs

Koajs et Node.js

Koajs est un framework web construit sur les bases de *Node.js*. Il permet la création d'API plus robustes et légères que celles construites uniquement à l'aide de son parent.

Node.js [7] est un *run-time* [13] asynchrone pour javascript basé sur un système d'événements. Normalement, javascript est un langage dit *interprété*. Cela signifie que pour exécuter un code écrit en javascript, un *interpréteur* va lire le code ligne par ligne et le transformer en instructions pour la machine. Un *run-time* permet une autre approche similaire à celle de java et la Java Virtual Machine (JVM). Dans ce cas le code est d'abord compilé en entier puis exécuté, ce qui améliore grandement les performances du langage.

[2] PUT, GET, POST, DELETE dans le cas d'HTTP. Plus d'informations *ici*

Le système d'événement au coeur de *Node.js* illustré dans la figure 3.6 fonctionne de la sorte. Le système maintient deux composants principaux, une queue d' *events* et une *thread pool*. Chaque requête envoyée au serveur est ajoutée à la queue, les événements sont traités dans une boucle infinie qui va prendre le premier élément de la queue et le traiter. *Node.js* différencie deux types d'événements :

— **Blocking** : Le code est exécuté de manière synchrone.

— **Non-Blocking** : Le code est exécuté de manière asynchrone.

Prenons le cas d'une requête vers une base de données, avec un *blocking event* étant donné que le système doit attendre que le code synchrone s'exécute, aucun autre événement ne peut être traité durant ce temps. C'est là qu'est utile la *thread pool* qui permet de traiter cet événement dans un autre thread le temps qu'il s'exécute et retourner ensuite le résultat. Les *non-blocking events* peuvent être exécutés dans la boucle principale car il ne font pas de *busy waiting* lors de l'attente de résultat et donc la boucle principale est libre de traiter d'autres événements tant que ceux-ci ne sont pas terminés, ayant juste à renvoyer le résultat au client une fois qu'elle a reçu une réponse asynchrone.

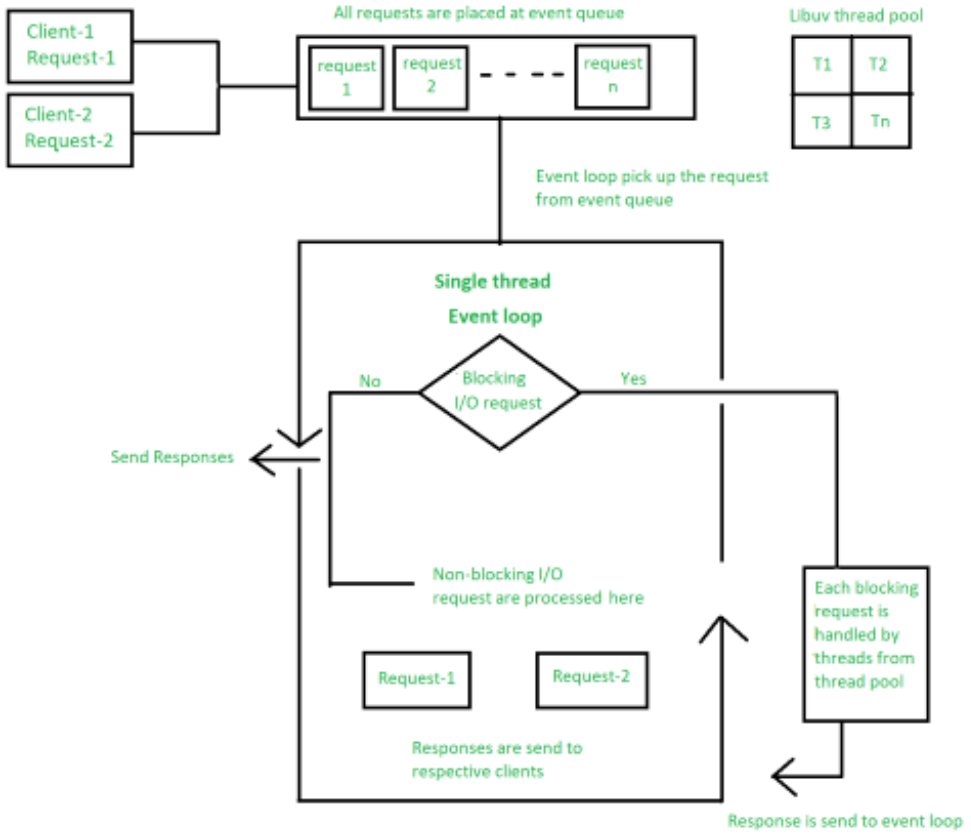


FIGURE 3.6. – Fonctionnement de Node.js, image tirée de `https://www.geeksforgeeks.org/explain-the-working-of-node-js/`

Koajs[5] utilise ce système mais standardise les requêtes dans un *contexte* (fig. 3.7). Ce *contexte* encapsule chaque requête d'un client et sa réponse du serveur dans un seul objet. Ainsi, le client et le serveur peuvent se l'échanger et le traiter de manière similaire.

Cela permet aussi l'ajout de *middlewares* qui vont pouvoir lire et traiter les requêtes indépendemment du fait qu'il s'agisse d'une requête ou d'une réponse. Concrètement ils peuvent ainsi gérer la sécurité et testant les droits d'accès du client avant même que le serveur ait à traiter la requête ou même générer la documentation de l'API.

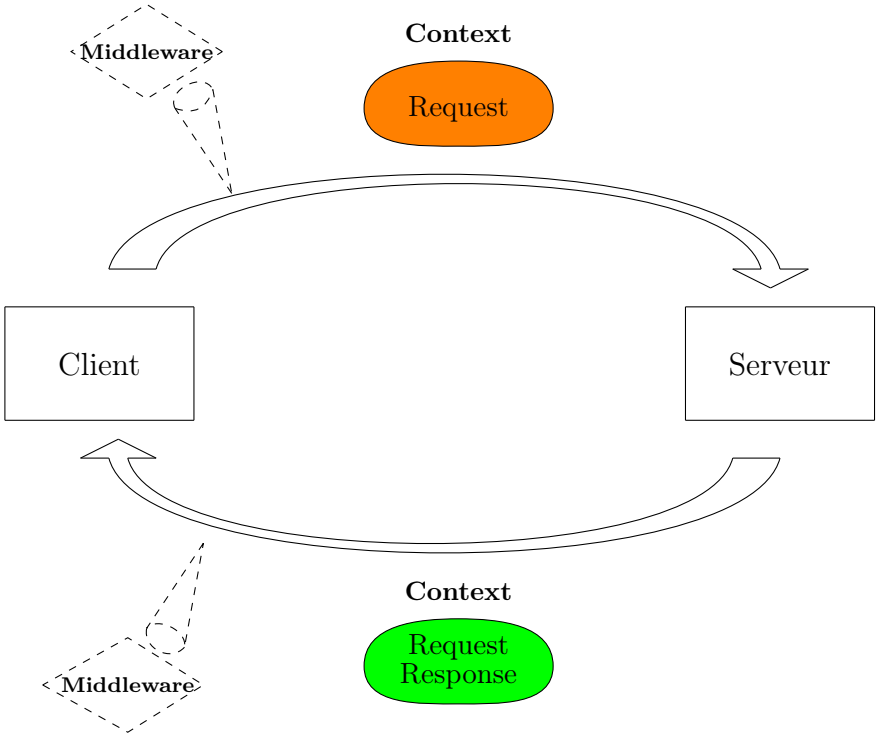


FIGURE 3.7. – Échange client-serveur dans Koajs

Swagger

La documentation de l'API (fig. 3.8) est générée automatiquement par un *middleware* comme présenté au point précédent. Cette documentation utilise l'outil *Swagger* pour offrir une interface donnant la possibilité de consulter ses fonctionnalités ainsi que de les tester. Ceci est particulièrement utile durant le développement de l'API car cela permet de tester son implémentation directement avec une mise en forme claire plutôt que de créer des requêtes depuis un terminal de commande.

L'utilisation de *Swagger* assure aussi de respecter les principes présentés dans la partie sur les services REST car cet outil garantit que la spécification *OpenAPI*[8] est respectée. Brièvement, cette spécification définit un standard pour la description d'API REST utilisant HTTP peu importe le langage de programmation utilisé.

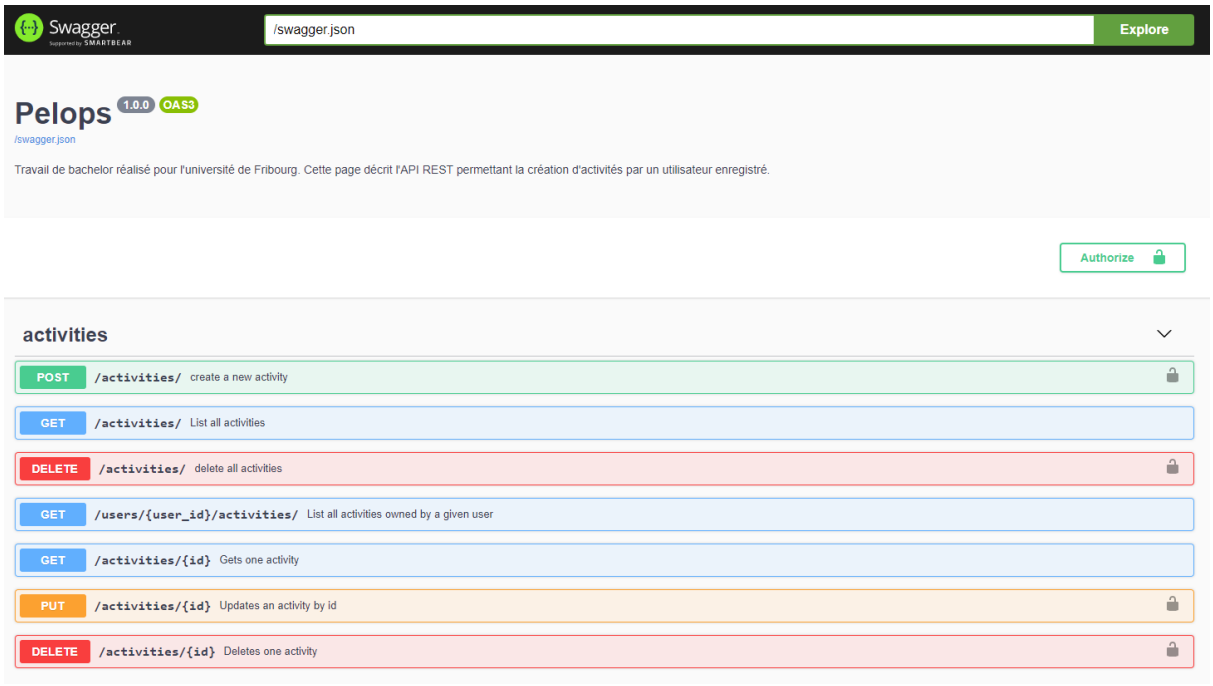


FIGURE 3.8. – Extrait de la documentation Swagger de l'API REST

3.3.3. Authentification

Comme mentionné précédemment, il est nécessaire de protéger certaines opérations de l'API. Si n'importe quel utilisateur pouvait supprimer le compte d'un autre cela mènerait vite à la disparition du service. C'est pourquoi l'on requiert un système d'authentification.

Dans ce projet, à la création de son compte un utilisateur définit un mot de passe qui sera crypté et stocké dans la base de donnée de manière à ce que même un administrateur du système soit dans l'impossibilité de connaître le vrai contenu du mot de passe. A sa connexion, l'utilisateur reçoit un JSON Web Token (JWT) qu'il joindra ensuite à chacune de ses requêtes nécessitant une authentification. Sans ce token, il lui est impossible de faire ces requêtes (fig. 3.9).

JSON Web Token

JWT [3] [4] est une manière compacte et cryptée de représenter des affirmations entre deux parties, par exemple une autorisation d'accès. Dans notre cas nous n'utilisons que la partie signature de JWT, appelée JSON Web Signature (JWS). Chaque signature a la structure suivante :

{header}.{payload}.{signature}

Où chaque partie correspond à :

- **header** : Contient un objet avec le nom de l'algorithme de cryptage du secret ainsi que le type de token.
- **payload** : Contient les données à échanger.

[3] Pour une présentation plus complète de JWT, la documentation complète de JWT est disponible *ici*

- **signature** : Contient une vérification de la signature. Cette vérification est faite en encodant l'*header* et le *payload* ainsi qu'un mot secret connu uniquement du créateur du système afin d'empêcher l'inversion du cryptage.

Chaque JWS est chiffré en utilisant l'algorithme base64url [4]. A noter qu'il est aussi possible d'utiliser JWT pour encrypter des données. Dans ce cas l'on utilise JSON Web Encryption (JWE), mais comme cette fonctionnalité n'est pas utilisée dans ce projet elle ne sera pas présentée plus en détail.

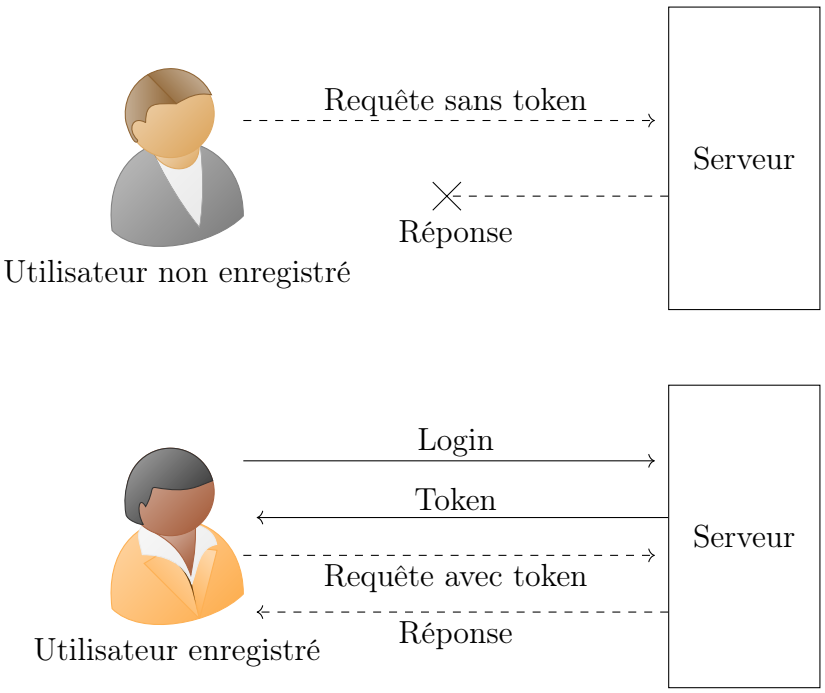


FIGURE 3.9. – Authentification et échange entre clients-serveur

3.3.4. Client mobile

L'application mobile a elle été réalisée avec le *Framework React Native* [10] développé par Facebook. Ce *Framework* permet de créer des applications sans se soucier du système d'exploitation du smartphone qui va l'utiliser. Ce qui permet de développer en parallèle pour IOS ou Android avec le même langage de programmation et la même structure de code. Sans cela, il faudrait écrire le code de l'application en Swift par exemple pour IOS et recommencer de zéro en Java ou Kotlin s'il on veut porter l'application vers Android.

Similairement à Hypertext Markup Language (HTML) et XML, *React Native* utilise un système de balises afin de créer des composants qui seront affichés à l'écran. Il est possible de créer ses propres balises, appelées composants, afin d'améliorer la lisibilité du code ainsi que sa réutilisabilité.

Un exemple basique de composant est visible dans le listing 3.3. A noter que le *framework* propose deux manières de structurer les composants, via des classes ou des fonctions. Leur fonctionnement est identique mais la plupart de la documentation récente semble favoriser l'approche fonctionnelle.

[4] Plus de détails sur le fonctionnement de base64url *ici*

```
import React from 'react';
import { Text } from 'react-native';
// Exemple fonctionnel
const Cat1 = () => {
  return (
    <Text>Hello, I am your cat!</Text>
  );
}
// Exemple avec classe
class Cat2 extends Component {
  render() {
    return (
      <Text>Hello, I am your cat!</Text>
    );
  }
}
export default {Cat1,Cat2};
```

Listing 3.3 – Code adapté d'un exemple tiré de la documentation de React Native [10]

La conversion entre le code javascript et celui de la plateforme hôte se fait grâce à *Fabric*, le système de rendu de *React Native*. Ce système écrit en C++ est responsable de prendre le code en javascript écrit par le programmeur et de le "traduire" dans le langage adapté au système d'exploitation de l'hôte.

Ce processus de rendu se fait en trois étapes illustrées dans la figure 3.10 :

1. **Render phase** : Le système va parcourir le code javascript et enregistrer chaque élément contenu dans un composant à l'intérieur d'un arbre.
2. **Commit phase** : Le système enregistre la position et la taille de chaque élément et indique que le nouvel arbre créé est le prochain à être affiché à l'écran.
3. **Mount phase** : Le système calcule les différences entre l'arbre courant et le prochain afin de définir les changements à appliquer aux composants rendus à l'écran. Le prochain arbre est ensuite affiché sur le smartphone.

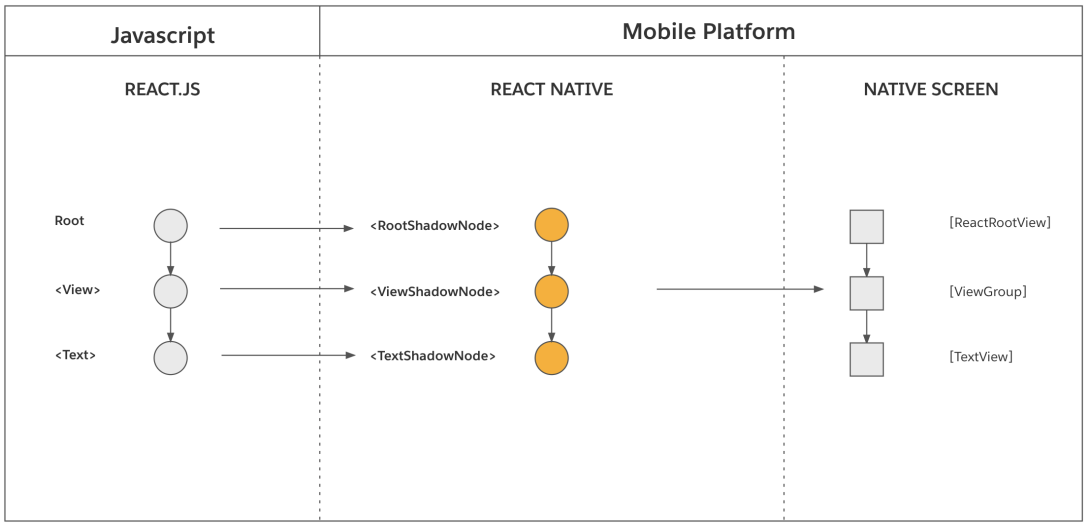


FIGURE 3.10. – Différentes étapes de rendu dans React Native (image tirée de la documentation[10])

Tester l'application

Pendant le développement de l'application, il est utile voir nécessaire de pouvoir visualiser les composants créés dans le code. Pour ce faire, on peut utiliser l'Integrated Development Environment (IDE) *Android Studio* qui offre la possibilité d'avoir une machine virtuelle de smartphone avec le système d'exploitation Android (fig. 3.11).

Afin de partager le code avec cette machine virtuelle, on utilise *Expo*. *Expo* est une plateforme de développement pour applications *React*. Cette plateforme peut accéder au port sur lequel est connectée la machine virtuelle et ainsi lui partager ce qui doit être rendu à l'écran comme si le smartphone possédait l'application. *Expo* existe aussi sous la forme d'application pour smartphone qui permet de tester le code directement sur son téléphone s'il est connecté au même réseau local que l'ordinateur sur lequel on l'écrit.

FIGURE 3.11. – Émulation de smartphone Google Pixel 2

4
Implémentation

4.1. Overview

Ce chapitre traite de l'aspect technique du projet. On y présente une partie des éléments de programmation clés utilisé dans le développement de l'application ainsi que de l'API REST. Plus précisément, la première partie décrit la génération de la documentation et la gestion des requêtes dans lAPI REST. La seconde présente la création de composants personnalisables dans *React Native* ainsi que l'organisation de la navigation entre les écrans de l'application.

4.2. API REST

Cette partie décrit une sélection d'éléments clés de l'implémentation de l'API REST. Pour les deux parties de cette section, nous allons nous concentrer sur la méthode GET de HTTP pour une liste des activités.

4.2.1. Générer la documentation

Afin de générer la documentation de l'API, il faut en premier lieu créer le *middleware Swagger*. Pour ce faire, on crée dans le listing 4.1 à la ligne 25 un objet *Swagger* avec les options spécifiées plus haut. Dans ces options on retrouve une définition de l'API et surtout, dans le champs *apis*, le chemin d'accès vers les codes responsables de gérer les requêtes.

```
1  const swaggerOptions = {
2      definition: {
3          openapi: '3.0.0',
4          info: {
5              title: 'Pelops',
6              version: '1.0.0',
7              description: 'Travail de bachelor realise pour l'universite de Fribourg.
                   Cette page decrit l'API REST permettant la
8              creation d'activites par un utilisateur enregistre.',
9          },
10     },
11     /**
12      * Paths to the API docs. The library will fetch comments marked
13      * by a @swagger tag to create the swagger.json document
14      */
15     apis: [
16         './controllers/activities.js',
17         './controllers/users.js',
18         './controllers/auth.js',
19     ],
20     // where to publish the document
21     path: '/swagger.json',
22 }
23
24 // Call our own middleware (see in file)
25 const swagger = Swagger(swaggerOptions);
```

Listing 4.1 – Création du middleware Swagger

Dans ces fichiers, on ajoute au code des commentaires commençants par l'annotation "@swagger" (listing 4.2). Dans ceux-ci, on spécifie principalement :

- **L'URI de la ressource**
  Ligne 4 : */activities/* ou ligne 17 : */users/user_id/activities/*
- **Le type de requête HTTP**
  Lignes 5 et 18 : *get*
- **Les potentiels paramètres de la requête**
  Lignes 23-29 : Ici l'on veut la liste des activités créées par un utilisateur, dont on doit spécifier l'identifiant. Le paramètre doit avoir un nom, une description et un type. On note aussi que le paramètre doit se retrouver à l'intérieur de l'URI en écrivant *in : path*.
- **Les réponses possibles et leur signification**
  Les codes d'erreur respectent les standards HTTP [1]. Lignes 9-15 : 200 signifie un succès et va retourner un schéma (fig. 4.1) similaire aux modèles présentés dans la partie sur *MongoDB*. Lignes 39-40 : On peut voir ici une explication supplémentaire au code 404 indiquant que l'identifiant fourni n'est relié à aucun utilisateur.
- **Une demande d'authentification**
  Vu que la lecture n'est pas une opération dangereuse, il n'est pas nécessaire d'être enregistré pour faire ces requêtes. On pourrait ajouter au commentaire *security : - bearerAuth :* [] pour exiger une authentification.

[1] Voir la liste des codes de réponse HTTP *ici*

En récoltant ces commentaires, un interface web est générée automatiquement par le *middleware* comme on peut le voir dans la figure 3.8 présentée précédemment.

```
/**
 * @swagger
 *
 * /activities/:
 * get:
 * summary: List all activities
 * tags:
 * - activities
 * responses:
 * '200':
 * description: success
 * content:
 * application/json:
 * schema:
 * $ref: '#/components/schemas/ActivitiesArray'
 *
 * /users/{user_id}/activities/:
 * get:
 * summary: List all activities owned by a given user
 * operationId: listUserActivities
 * tags:
 * - activities
 * parameters:
 * - name: user_id
 * in: path
 * required: true
 * description: the id of the owner
 * schema:
 * type: string
 * responses:
 * '200':
 * description: success
 * content:
 * application/json:
 * schema:
 * type: array
 * items:
 * $ref: '#/components/schemas/Activity'
 * '404':
 * description: User not found
 */
```

Listing 4.2 – Commentaires ajoutés au code afin de générer la documentation

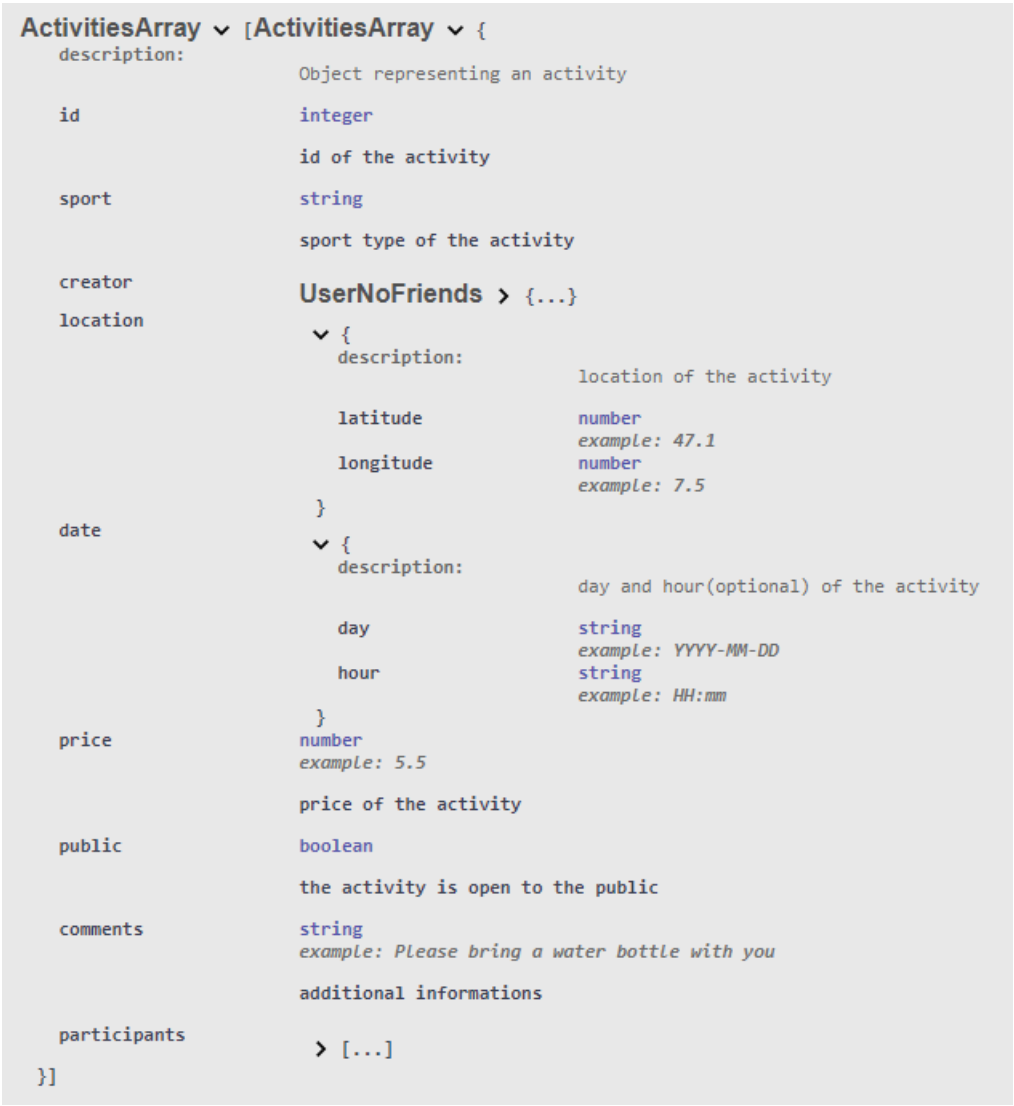


FIGURE 4.1. – Schéma pour une liste d'activités

4.2.2. Gérer une requête

Lorsqu'un client adresse une requête au serveur, il doit être en mesure de répondre adéquatement. Il doit être au courant des différentes requêtes qu'il peut recevoir. Pour ce faire on utilise un *routeur* qui enregistres ces différentes possibilités. On voit dans le listing 4.3 un exemple pour les routes liées aux activités. On peut y lire le type de requête HTTP, l'URI de la ressource touchée, le besoin d'authentification et finalement la méthode exécutée lors de la requête.

```
1
2 const activities = require('../controllers/activities');
3 const jwt = require('../middlewares/jwt');
4
5 module.exports = (router) => {
6     router
7         .param('activity_id', activities.getById)
8         .post('/activities/', jwt, activities.create)
9         .get('/activities/', activities.list)
```

```
10        .put('/activities/:activity_id', jwt, activities.update)
11        .delete('/activities/:activity_id', jwt, activities.delete)
12        .get('/activities/:activity_id', activities.read)
13        .delete('/activities/', jwt, activities.clear);
14
15  }
```

Listing 4.3 – Ensemble des routes liées aux activités

Pour reprendre la méthode GET présentée dans la partie précédente, on voit que la méthode appelée est la méthode *list*, observable dans le listing 4.4.

Comme mentionné précédemment, cette méthode traite à la fois du cas où l'on veut recevoir l'entier de la liste des activités et celui où l'on veut uniquement celles créées par un utilisateur dont on fournit l'identifiant. Cette différence est prise en compte aux lignes 3 et 4 où une requête vers la base de donnée est effectuée. On lui demande de trouver (*Find()*) des activités, avec en paramètre *req* qui peut avoir deux valeurs :

— Un objet vide {} : Aucun utilisateur n'a été spécifié, la recherche n'est donc pas limitée et *req* garde sa valeur initiale.

— Un objet contenant {creator : user._id} : Un utilisateur a été spécifié dans la requête, on va créer un champs *creator* dans *req* et lui assigner l'identifiant de celui-ci. La recherche est ensuite limitée aux activités dont le champs *creator* est identique à celui de *req*.

Ces données sont ensuite *peuplées*. En effet, afin d'optimiser le stockage, les champs faisant référence à d'autres modèles ne contiennent qu'un identifiant. En les peuplant, on va chercher l'entier du modèle référencé ce qui permet par exemple de connaître aussi le nom du créateur d'une activité.

Finalement, on appelle la méthode *toClient()* sur chacun des objets retournés. Cette méthode permet de transformer les données contenues dans la base de données en données plus présentables, en supprimant par exemple des champs utilisés uniquement dans la base de donnée comme l'heure de création ou alors pour des raisons de sécurité des champs que l'on ne souhaite pas partager comme un mot de passe.

```
1    list: async (ctx) => {
2        const req = {};
3        if (ctx.user) req.creator = ctx.user._id;
4        let activities = await Activity.find(req)
5            .populate('creator')
6            .populate('participants')
7            .exec();
8        for (let i = 0; i < activities.length; i++) {
9            activities[i] = activities[i].toClient();
10       }
11       ctx.body = activities;
12   },
```

Listing 4.4 – Méthode retournant une liste d'activités

4.3. Application mobile

Cette partie traite de l'implémentation du client mobile décrit quelques éléments clés de celle-ci, comment créer un composant ainsi que la gestion de la navigation entre les écrans.

4.3.1. Créer un composant modifiable

React Native met à disposition tout un ensemble de composants standards qui peuvent être importés dans le code sans modification particulière. Similairement au développement web "traditionnel" utilisant HTML pour la structure des éléments avec Cascading Style Sheet (CSS) pour leur donner un aspect plus plaisant, *React Native* inclut des feuilles de style pour modifier l'aspect des composants. Cependant, pour ces composants standards, les possibilités de modification sont limitées et s'adaptent à la plateforme sur laquelle ils sont affichés (fig. 4.2).

(a) Boutons sur IOS

(b) Boutons sur Android

FIGURE 4.2. – Adaptation des boutons pour un même code, images tirées de la documentation [10]

Afin de pallier à ceci, il est possible de créer ses propres composants, laissant le choix libre quant aux limitations des possibilités de modification. Afin d'illustrer ce processus, le listing 4.5 contient le code nécessaire à la création d'un bouton modifiable.

Chaque composant créé par le développeur n'est en fait qu'un assemblage de composants de base fournis par *React Native*. On voit ligne 2 la clause d'importation de ceux utilisés pour la création du bouton.

- **Stylesheet** est la classe de *React Native* utilisée pour créer des feuilles de styles comme à la ligne 15.
- **Text** est la balise indiquant le début d'un texte à afficher à l'écran.
- **View** est le composant le plus basique. Cette balise indique simplement le début de contenu.
- **TouchableOpacity** est la balise indiquant le début d'une zone que l'utilisateur peut presser pour interagir avec l'application.

Ligne 5 on trouve la déclaration de la fonction représentant le composant qui prend comme paramètre un objet contenant des variables :

- **onPress** contient une référence vers une fonction à exécuter lorsque le bouton est pressé.
- **title** contient le titre que doit porter le bouton.
- **sytle** contient l'aspect du bouton. On peut voir un exemple de style lignes 16 à 23.
- **textStyle** contient le formatage du titre du bouton.

Cette liste est non-exhaustive et correspond uniquement au besoins du développeur. On voit que certaines de ces variables peuvent être laissées vides afin d'utiliser des valeurs par défaut. C'est pourquoi des tests sont effectués lignes 7 et 9 afin de savoir si l'utilisateur de ce composant souhaite les modifier.

On peut observer dans le listing 4.6 l'utilisation de ce composant. On note que le style est modifié alors que le format du texte lui est gardé par défaut et qu'une fois pressé, ce bouton doit appeler la fonction *register*.

```
1  import React from 'react';
2  import { StyleSheet, Text, View, TouchableOpacity } from 'react-native';
3  import { colors } from '../styles/global';
4
5  export default function CustomButton({onPress, title, style, textStyle}){
6      return(
7          <View style={style == null ? styles.button : style}>
8              <TouchableOpacity onPress={onPress}>
9                  <Text style={textStyle == null ? styles.buttonText : textStyle}>{title
                       }</Text>
10             </TouchableOpacity>
11         </View>
12     );
13 }
14
15 const styles = StyleSheet.create({
16     button:{
17       backgroundColor: colors.buttonsBackgroundLight,
18       padding:10,
19       marginHorizontal:'30%',
20       alignItems: 'center',
21       justifyContent: 'center',
22       borderRadius:20,
23       borderWidth:1.5
24     },
25     buttonText:{
26         fontSize: 16,
27         color: colors.textDark,
28     }
29   });
```

Listing 4.5 – Code pour la création d'un bouton modifiable

```
1          (<CustomButton
2            style={{
3              backgroundColor: colors.buttonsBackgroundLight,
4              padding:10,
5              alignItems: 'center',
6              justifyContent: 'center',
7              borderWidth:1.5,
8              borderColor: colors.textHighlight
9            }}
10           title='Inscription'
11           onPress={register}/>):
```

Listing 4.6 – Ajout d'un bouton à un écran

4.3.2. Navigation

Une application créée avec *React Native* est en fait un ensemble d'écrans. Chacun de ceux-ci peut renvoyer vers un autre lors d'interactions. La gestion de ces liens est gérée par la *navigation* intégrée au *framework*.

Le développeur crée des *piles* (*stack* en anglais) qui vont enregistrer les possibilités de navigation depuis un écran. Chaque pile contient un écran de départ sur lequel va arriver l'utilisateur initialement ainsi qu'une liste d'écrans atteignables. Par exemple dans le listing 4.7 le premier écran est l'écran d'accueil *Home*, ligne 6, et depuis celui-ci on peut naviguer vers les détails d'une activité, *ActivityDetails* ligne 12.

Depuis chaque écran de base, lorsque l'on navigue vers un autre écran, celui-ci vient se poser sur le précédent, d'où le nom de *pile*. Ainsi, il est possible de revenir en arrière simplement en dépilant le premier élément de la *stack* active.

```
1  export default function HomeNavigator({navigation}) {
2      return(
3          <Navigator mode='modal'>
4              <Screen
5                  name="Home"
6                  component={Home}
7                  options={{
8                      headerShown:false
9                  }}/>
10             <Screen
11                 name="Details"
12                 component={ActivityDetails}
13                 options={{
14                     title:'',
15                     headerStyle:{
16                         backgroundColor: colors.background,
17                         shadowColor:'transparent',
18                         elevation:0,
19                     },
20                     headerTintColor: colors.inactive,
21                 }}/>
22         </Navigator>
23     );
24 }
```

Listing 4.7 – Stack gérant les interactions depuis l'écran d'accueil

Ces piles peuvent aussi être combinées. En effet, l'application utilise plusieurs onglets (fig. 4.3) pour naviguer entre la carte et la liste des activités ou le profil d'utilisateur. Dans ce cas, on ne navigue pas entre des écrans directement mais bien entre des piles car chaque onglet contient sa propre navigation. On voit dans le listing 4.8 que les composants contenus dans chaque option sont des *Navigator* comme celui présenté dans le listing 4.7.

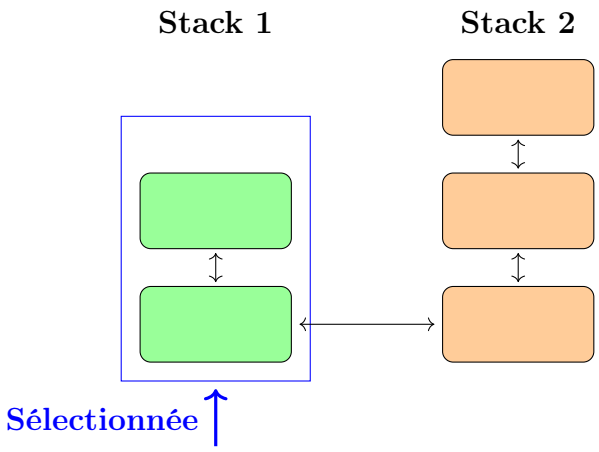
FIGURE 4.3. – Différents onglets de l'application

```
1            <Screen name='Activities' component={ActivitiesNavigator} />
2            <Screen name='Home' component={HomeNavigator} options={({navigation}) =>
3                ({ headerTitle: props => <Header navigation={navigation} {...props} />
                     })}
4                />
5            <Screen name='User Profile' component={AboutNavigator} />
```

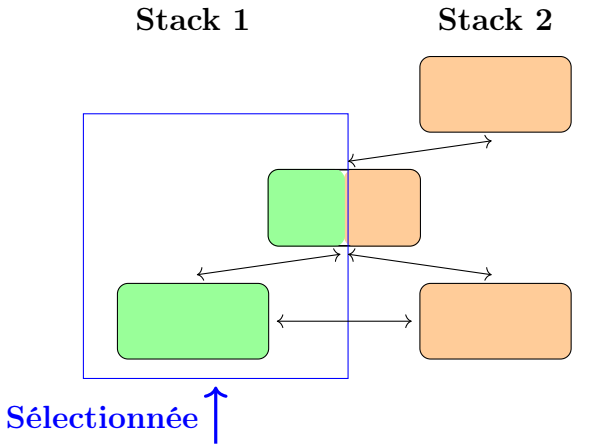
Listing 4.8 – Stack gérant les onglets de l'application

La figure 4.4 résume les possibilités de navigation et traite du cas où un écran est ajouté dans plusieurs piles. La pile sélectionnée est celle affichée à l'écran du smartphone. Il est possible de naviguer entre différentes piles en changeant d'onglet par exemple ou entre écrans à l'intérieur d'un même pile. Dans certains cas, un écran peut être présent dans plusieurs piles.

Un exemple tiré de l'application est l'écran contenant les informations sur une activité. Cet écran est atteignable depuis la carte principale ainsi que depuis la liste des activités et donc devra tenir compte de ceci. Si l'utilisateur fait un retour arrière depuis celui-ci, il sera redirigé vers la carte s'il avait ouvert les informations depuis cet écran et similairement vers la liste des activités dans l'autre cas.



(a) Navigation avec deux stacks



(b) Navigation avec deux stacks partageant un écran

FIGURE 4.4. – Navigation dans React Native

Partager des informations entre les écrans

Ce système des piles isolées permet l'échange d'information entre écrans d'une même pile mais rend la tâche plus compliquée lorsque des données doivent être partagées de manière plus globale. On pourrait par exemple mettre des préférences d'utilisateur comme un thème sombre dans cette catégorie.

Dans le cadre de l'application *Pelops*, on désire connaître le token reçu par l'utilisateur lors de sa connexion et son identifiant principalement. En effet, un token non nul permet de savoir si l'utilisateur a réussi sa connexion et ainsi de naviguer de l'écran de connexion vers l'écran principal. Son identifiant permet d'avoir une référence vers l'utilisateur lors de requêtes vers l'API.

Pour ce faire, on utilise un *contexte* (fig. 4.5). Ce contexte est stocké en parallèle du système de navigation et chaque écran peut y accéder sans limitation par la pile dans laquelle il est contenu.

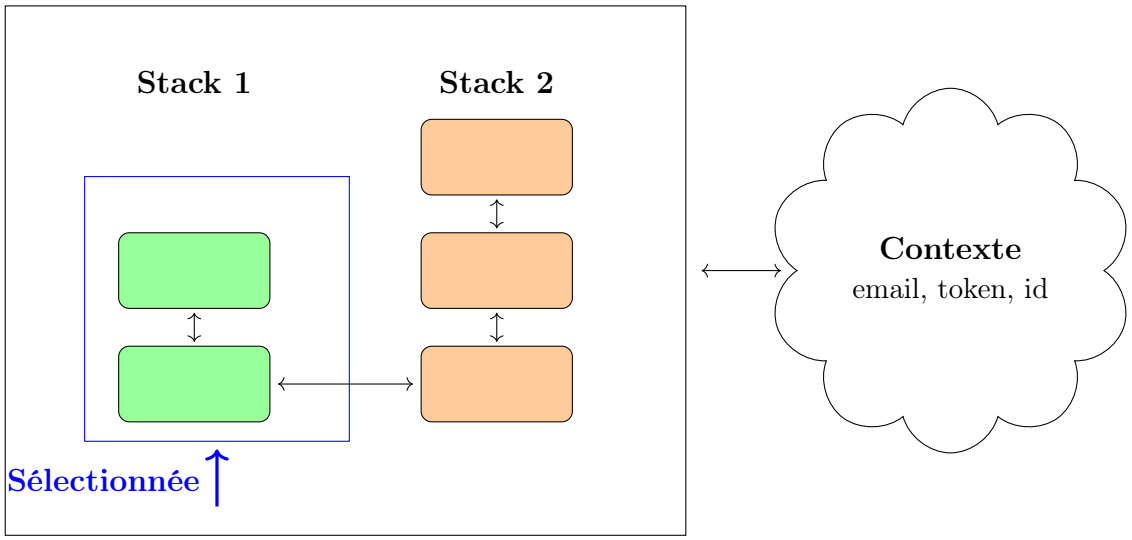


FIGURE 4.5. – Échange avec le contexte

5 Conclusion

5.1. En résumé

Ce rapport présente l'utilisation d'une API REST utilisant une base de donnée *MongoDB* pour stocker les données et un client mobile construit avec *React Native*. Le but de ceux-ci est de créer une plateforme sur laquelle des gens peuvent créer et rejoindre des activités sportives amateures en utilisant une carte interactive. Ce projet s'inscrit dans le cadre d'un réel projet appelé *Pelops* qui va continuer de se développer dans le futur.

5.2. Obstacles rencontrés et potentielles améliorations

Ce projet est ma première expérience dans le développement d'applications pour smartphone. C'est pourquoi il était parfois difficile d'estimer la complexité de l'implémentation d'un système ainsi que de réussir à limiter son envergure, chaque succès donnant envie d'ajouter des nouvelles fonctionnalités.

Comme mentionné dans la partie traitant de l'organisation du projet, ce travail a été réalisé durant la crise du Covid. Le travail a distance a impacté mon efficacité ainsi que ma motivation et a duré plus long qu'il ne l'aurait dû.

Ce projet reste un entrée dans le monde du développement et de nombreuses fonctionnalités doivent être ajoutées afin de faire de l'application un projet commercial sérieux. De plus, en situation réelle, la protection des données personnelles est un sujet sensible qui mérite plus de ressources que le minimum fourni dans le cadre de ce projet.

5.3. Derniers mots

Cette thèse de Bachelor était une introduction très intéressante dans le développement de projets à plus grande échelle. C'était aussi l'occasion d'apprendre de nouveau concepts seul sur la base des connaissances acquises durant mes études.

A

Acronymes communs

ANSI American National Standards Institute
API Application Programming Interface
CERN European Organization for Nuclear Research
CRUD Create, Read, Update, Delete
CSS Cascading Style Sheet
DBMS Database Management System
ERM Entity Relationship Model
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
IDE Integrated Development Environment
IoT Internet of Things
IP Internet Protocol
IPSec Internet Protocol Security
JSON JavaScript Object Notation
JVM Java Virtual Machine
JWE JSON Web Encryption
JWS JSON Web Signature
JWT JSON Web Token
RDBMS Relational Database Management System
RFID Radio Frequency Identification
REST Representational State Transfer
ROA Resource Oriented Architecture
SOA Service Oriented Architecture
SQL Structured Query Language
TCP Transmission Control Protocol
TSDBMS Times Series Database Management System
URI Unified Resource Identifier
URL Uniform Resource Locator
VPN Virtual Private Network
W3C World Wide Web Consortium
WADL Web Application Description Language
WSDL Web Service Description Language
WoT Web of Things
XML eXtensible Markup Language

B

License of the Documentation

C

Repository github du projet

Le repository github de ce projet est disponible à l'adresse : `https://github.com/a2mont/sport-with-me`

Sur celui-ci on peut trouver.

- Le code source du projet
- Une vidéo de démonstration de l'utilisation de l'application (cette vidéo est aussi accessible par ce lien : `https://youtu.be/CwIYeuPikgE`)
- Une marche à suivre pour l'exécution du code

Sources

[1] AdobeXD. https://www.adobe.com/ch_fr/products/xd.html (dernière consultation le Avril 20, 2022).

[2] Effet de réseau, Wikipedia. https://fr.wikipedia.org/wiki/Effet_de_r%C3%A9seau (dernière consultation le Mars 31, 2022).

[3] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (dernière consultation le Juillet 30, 2005).

[4] JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519 (dernière consultation le Avril 24, 2022).

[5] Documentation koajs. https://koajs.com/ (dernière consultation le Avril 21, 2022).

[6] What is MongoDB ? Introduction, Architecture, Features Example. https://www.guru99.com/what-is-mongodb.html (dernière consultation le Avril 21, 2022).

[7] Explain the working of Node.js. https://www.geeksforgeeks.org/explain-the-working-of-node-js/ (dernière consultation le Avril 21, 2022).

[8] OpenAPI Specification v3.1.0. https://spec.openapis.org/oas/latest.html (dernière consultation le Avril 22, 2022).

[9] Jacques Pasquier. Advanced software engineering course, 2021.

[10] React Native Documentation. https://reactnative.dev/ (dernière consultation le Avril 22, 2022). iii, 1

[11] React Native Navigation Documentation. https://reactnavigation.org/ (dernière consultation le Avril 24, 2022).

[12] Une API REST, qu'est-ce que c'est ? https://www.redhat.com/fr/topics/api/what-is-a-rest-api (dernière consultation le Avril 21, 2022).

[13] Google V8 Documentation. https://v8.dev/docs (dernière consultation le Avril 24, 2022).

[14] What Is Scrum Methodology ? Scrum Project Management. https://www.digite.com/agile/scrum-methodology/ (dernière consultation le Avril 20, 2022).

[15] Documentation swagger. https://swagger.io/ (dernière consultation le Avril 22, 2022).

[16] Trello. https://trello.com/fr (dernière consultation le Avril 20, 2022).