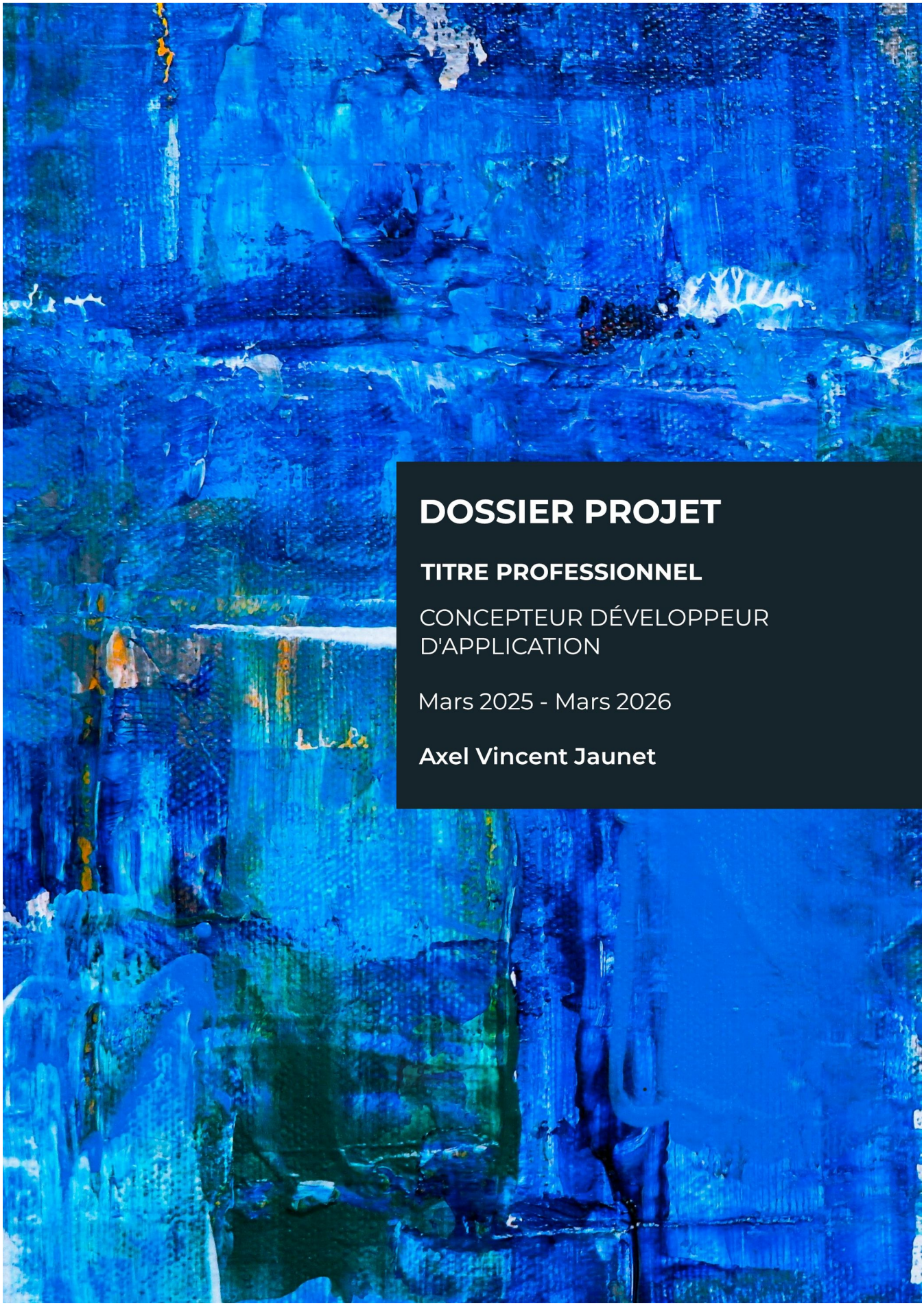


DOSSIER PROJET

TITRE PROFESSIONNEL

CONCEPTEUR DÉVELOPPEUR D'APPLICATION

Mars 2025 - Mars 2026

Axel Vincent Jaunet

Sommaire

Résumé

Durant ma formation de Concepteur Développeur d'Applications, j'ai réalisé mon alternance au sein du Laboratoire Lapsa. Bien que diverses tâches m'étaient confiées, je ne pouvais présenter de projet couvrant l'ensemble des différentes compétences du référentiel. Ainsi, sur mon temps de formation, j'ai réalisé un projet que j'avais eu l'idée quelques mois auparavant.

Après avoir longtemps réfléchi à ce projet, j'ai décidé de mettre au point une architecture en n-tiers, avec un client en Kotlin, compatible avec l'ensemble des appareils Android. L'objectif était de faire une application simple, intuitive et facile à utiliser.

Pour concrétiser mon projet, j'ai réalisé un cahier des charges et des spécifications techniques et fonctionnelles. Elles m'ont permis de poser les bases et d'anticiper plusieurs problèmes potentiels.

Afin de valider mon projet, j'ai mis en place des tests unitaires et fonctionnels avant de déployer l'application.

Présentation du projet

Sentinelle est née d'un constat personnel. À chaque fois que je sortais de chez moi, je me sentais obligé de prévenir un proche de mon trajet ou de mon arrivée, une habitude rassurante mais rapidement contraignante. Un jour, en me rendant à un rendez-vous incertain, une remise en main propre avec un inconnu via Leboncoin, j'ai réalisé à quel point cette précaution, bien que nécessaire, était fastidieuse à répéter. De plus, elle ne garantissait pas une réaction rapide en cas de problème réel.

Ce sentiment n'est pas unique. Beaucoup de gens, surtout dans les grandes villes ou lors de déplacements isolés, ressentent une appréhension à sortir seuls, que ce soit pour des trajets quotidiens, des retours tardifs ou des rencontres avec des inconnus. Cette peur, parfois irrationnelle mais bien réelle, peut limiter la liberté de mouvement et peser sur le quotidien.

C'est de cette frustration et de ce besoin de sécurité simplifiée que m'est venue l'idée : et si une application pouvait automatiser cette vigilance ? Plutôt que d'envoyer manuellement un message à chaque déplacement, Sentinelle permet de définir un minuteur. Si je n'annule pas ce compte à rebours à mon arrivée (ou si un incident survient), mes proches reçoivent automatiquement ma localisation et des enregistrements audio, sans que j'aie à faire quoi que ce soit. Ainsi, je peux me déplacer sereinement, même dans des situations potentiellement risquées, en

sachant que mes proches seront alertés seulement si nécessaire.

Au-delà de mon usage personnel, ce système répond à un besoin universel : allier sécurité et simplicité, pour que chacun puisse vaquer à ses occupations sans la charge mentale de devoir constamment rassurer son entourage. Sentinelle n'est pas qu'une application, c'est la concrétisation d'une solution pensée pour ceux qui, comme moi, refusent de choisir entre liberté et sécurité.

Liste des compétences couvertes par le projet

La conception de ce projet a pu me permettre de répondre à l'entièreté des compétences énumérées dans le référentiel de certification du titre professionnel de Concepteur Développeur d'Applications. Voici la liste des compétences couverte par ce projet :

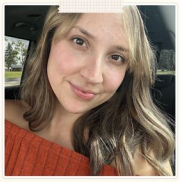
- Maquetter une application
- Développer une interface utilisateur de type desktop
- Développer des composants d'accès aux données
- Développer la partie « front-end » d'une interface utilisateur web
- Développer la partie « back-end » d'une interface utilisateur web
- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données
- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
- Concevoir une application
- Développer des composants métier
- Construire une application organisée en couches
- Préparer et exécuter les plans de tests d'une application
- Préparer et exécuter le déploiement d'une application

Cibles

Sentinelle s'adresse à des populations variées, exposées à des risques spécifiques dans leur quotidien :

- Femmes se déplaçant seules la nuit.
- Minorités persécutées.
- Écoliers victimes de harcèlement ou de racket.
- Prévention en cas d'accident de la route.
- Sportifs et randonneurs.

- Personnes atteintes de troubles cognitifs.
- Travailleurs isolés.



Nina Tention

- 27 ans
- Coach en self contrôle
- Limoges
- Fais du Yoga
- Adore le mojito

Bio

Nina Tention, 27 ans, est une jeune femme maladroite et tête en l'air. Chaque vendredi et dimanche soir, elle passe trois heures sur la route pour rendre visite à sa famille. Elle a déjà eu plusieurs petits accrochages à cause de sa distraction, et elle a tendance à renverser ses boissons sur ses vêtements, surtout quand elle est pressée. Elle oublie fréquemment ses clés, ce qui la fait revenir chez elle en panique. En dehors de ses mésaventures quotidiennes, elle adore le yoga et la lecture, même si elle n'a jamais réussi à finir un livre.

Emilie Jentremble

- 15 ans
- Lycéenne
- Nantes
- Skateuse
- Aime les chats

Bio

Emilie étudie au lycée Clémenceau à Nantes. Chaque jours elle se rend au lycée à pied. Le soirs, elle passe des heures au skateparc pour tenter des figures assez extrême.

Indépendante, un peu tête brûlée, Émilie ne veut pas dépendre des autres pour sa sécurité. Elle veut pouvoir se gérer. Se sentir forte, sans pour autant se mettre en danger.

Eva Cuation

- 51 ans
- Plombier
- Saint-Pierre-en-Faucigny
- Chasseuse

Bio

Eva Cuation, 51 ans, est plombier et travaille souvent seule, ce qui la rend parfois vulnérable. Elle intervient parfois chez des gens pas forcément nets, et certaines situations peuvent la mettre en insécurité. Malgré cela, Eva reste indépendante et pragmatique, toujours prête à faire face à l'inconnu. En dehors de son travail, elle est une passionnée de chasse et trouve son équilibre dans la nature. À 51 ans, elle est aguerrie, et bien qu'elle se sente parfois mal à l'aise dans certaines interventions, elle n'hésite jamais à foncer, un kit d'outils et son fusil en main.

Khaled Réziste

- 28 ans
- Infirmier
- Paris
- Musicien amateur
- Musulman

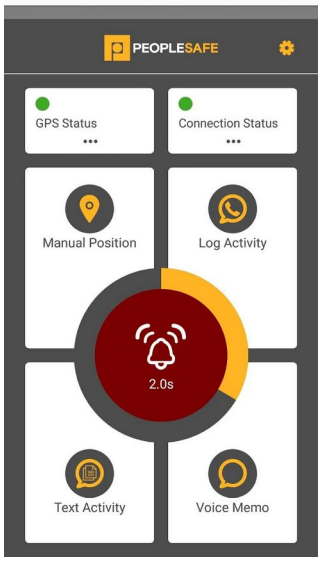
Bio

Khaled, 28 ans, infirmier à Paris et musicien amateur, fait face à des agressions verbales dans la rue en raison de ses origines. Mais plutôt que de répondre à la provocation, il préfère garder son calme. Avec son sens aigu de la justice, il utilise discrètement des enregistrements sonores pour prouver les insultes et dénoncer l'injustice. Pour Khaled, chaque agression est une occasion de se battre pour ses droits sans céder à la haine.

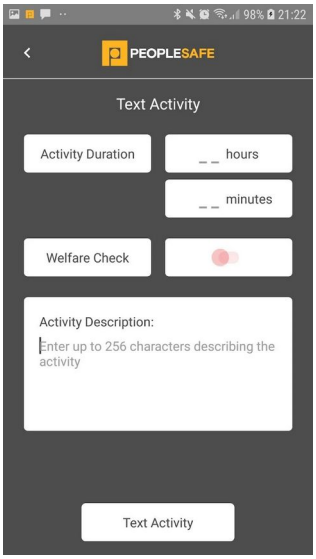
Veille concurrentielle

App-elles est une application mobile, créée en 2015 par Résonante en 2015. Résonante est une association à but non lucratif qui permet d'alerter ses proches lorsque l'on se sent en danger. Elle permet de mettre en relation plusieurs proches qui peuvent écouter ce qu'il se passe. L'appli peut également appeler des

secours. Les points négatifs de l'application : n'est pas intuitive et pour comprendre le fonctionnement, il faut se rendre sur le site web. Les proches "'protecteur" doivent forcément installer l'application.



PeopleSafe, propose un bouton SOS qui lance une connexion bi-directionnel avec un centre de réception d'alarme. L'alarme peut se déclencher toute seule en cas de chute. PeopleSafe propose lui aussi un minuteur et lorsqu'il se déclenche, préviens les secours. Cependant, elle n'enregistre aucune donnée pendant que le minuteur fonctionne. Cette application est destinée aux travailleurs isolés.



Ces applications permettent de déjouer des agressions mais ne permettent pas de tracer les personnes en cas d'enlèvement ou dans des situations où l'utilisateur n'a pas eu le temps de déclencher le SOS (Accident de la route, malaise, enlèvement …)

Technologie Utilisé

Le projet est constitué de 2 éléments principaux : un site web et une application mobile.

Site web

Le site web est conçu avec Django, un framework python. Une vitrine est dessus mais il ne sert qu'essentiellement pour l'API.
Sur cette API, est utilisée :

- une base de données Mysql
- le service Twilio pour envoyer des SMS
- Firebase pour l'authentification

Application Mobile

Réalisé en Kotlin, avec Jetpack Compose, un langage Natif créé par Google ce qui

permet des performances meilleures qu'un langage cross-platform.
La base de données sera accessible depuis l'application mobile grâce à une API qui sera développée et intégrée au site web.

Site Web : Fonctionnalités attendues

Fonctionnalités obligatoires

Le site servira uniquement pour le visiteur qui reçoit un lien d'un utilisateur. Il pourra consulter les enregistrements de son proche et les télécharger. Il aura également la possibilité de contacter le développeur si besoin d'information complémentaire.

Application Mobile : Fonctionnalités attendues

Pour les visiteurs :

Fonctionnalités obligatoires

- Se créer un compte utilisateur
- Se connecter à son compte utilisateur
- Consulter un trajet
- Envoyer un message via le formulaire de contact

Pour les utilisateurs :

Fonctionnalités obligatoires

- Renseigner un numéro de contact
- Définir un message personnalisé
- Définir un retardateur
- Enregistrer la position géographique
- Enregistrer le micro
- Envoyer un sms à un contact
- Modifier ses informations personnelles.
- Supprimer les enregistrements
- Supprimer ses données d'un clic
- Bouton SOS (Envoie la position GPS actuel à un contact)

Gestion de projet

Afin de gérer efficacement et organiser mon projet, j'ai décidé d'appliquer le modèle « cycle en v ».

Son intitulé « cycle en v » se justifie par la composition de ses trois phases et des communications possibles entre les différentes étapes qui les composent (voir figure 1). Ces trois phases sont :

- la phase descendante dans laquelle on étudie les besoins du client et réfléchit à comment y répondre.
- La phase de réalisation du projet (codage)
- La phase ascendante qui permet de vérifier et valider le projet.

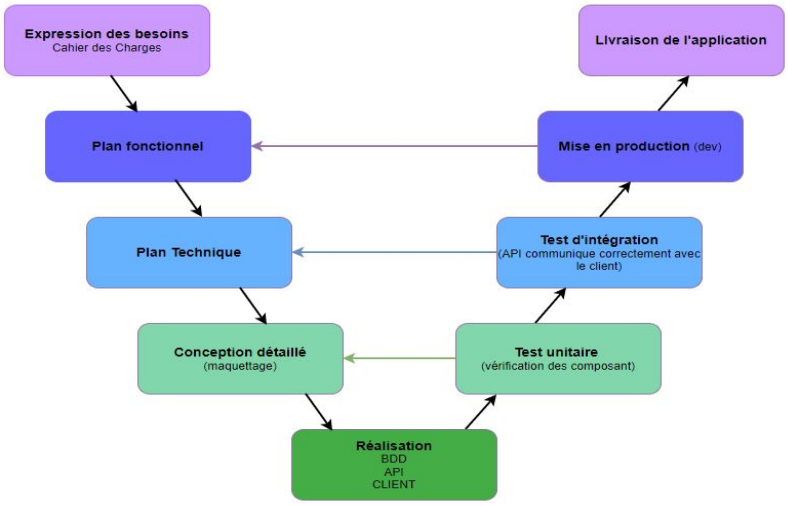


Figure 1: Diagramme du cycle en v

J'ai également fait la liste des fonctionnalités à faire, ainsi qu'un planning pour savoir combien de temps ça me prendrait.

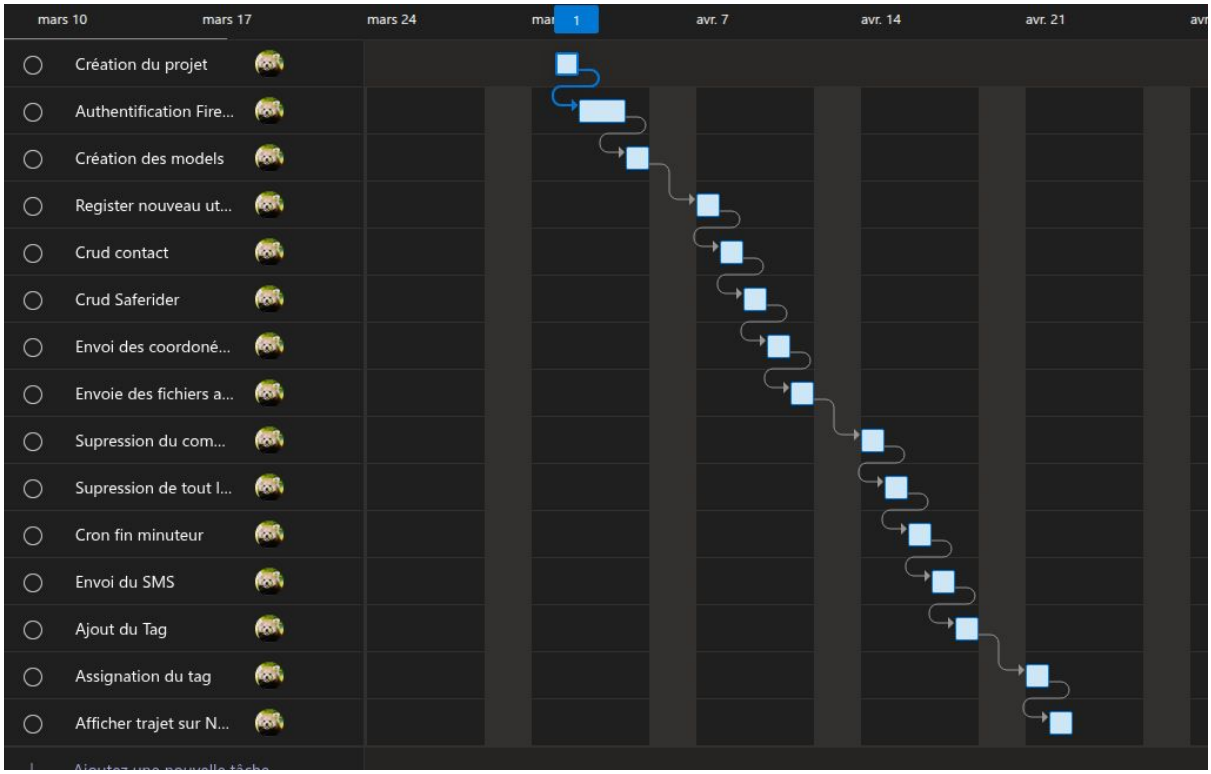


Figure 2: Chronologie de la conception du projet

Nom de la tâche	Début	Dépe...	Attribu...	Du...
Création du projet	1/4/2025		Axel	1 jour
Authentification Firebase	2/4/2025	1FS	Axel	2 jours
Création des models	4/4/2025	2FS	Axel	1 jour
Register nouveau utilisateur	7/4/2025	3FS	Axel	1 jour
Crud contact	8/4/2025	4FS	Axel	1 jour
Crud Saferider	9/4/2025	5FS	Axel	1 jour
Envoi des coordonées GPS	10/4/2025	6FS	Axel	1 jour
Envoie des fichiers audio	11/4/2025	7FS	Axel	1 jour
Supression du compte et des données	14/4/2025	8FS	Axel	1 jour
Supression de tout les trajets	15/4/2025	9FS	Axel	1 jour
Cron fin minuteur	16/4/2025	10FS	Axel	1 jour
Envoi du SMS	17/4/2025	11FS	Axel	1 jour
Ajout du Tag	18/4/2025	12FS	Axel	1 jour
Assignation du tag	21/4/2025	13FS	Axel	1 jour
Afficher trajet sur Navigateur	22/4/2025	14FS	Axel	1 jour

Figure 3: Liste des tâches

Spécification Fonctionnelle

À partir du besoin, j'ai établi 2 acteurs principaux : L'utilisateur, et le contact de l'utilisateur. Il y a également un troisième acteur qui est l'administrateur.
Pour ce qui est des deux acteurs principaux, ils n'interagissent pas de la même façon avec l'application. En effet l'utilisateur va utiliser le client mobile, tandis que son contact va utiliser le site web.

Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation, également appelé "use case", permet de représenter visuellement les fonctionnalités de chaque acteur.

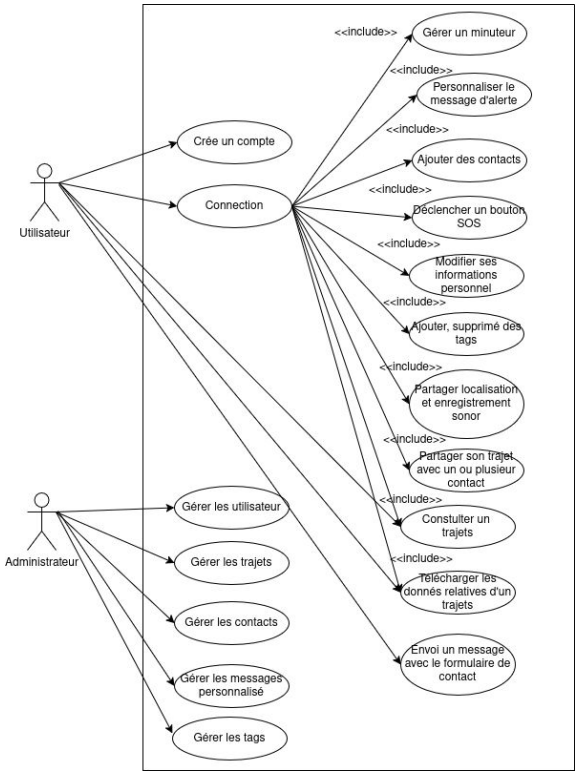


Figure 4: Diagramme de cas d'utilisation

Diagramme d'activité

Pour la fonctionnalité la plus compliquée de mon projet, j'ai décidé de faire un diagramme d'activité pour construire un algorithme et bien assimiler les actions étapes par étapes.
La fonctionnalité décrite ci-dessous, correspond à la récupération des données GPS, et audio lors de l'envoie sur un serveur.

Il arrive parfois que la connexion à internet soit interrompue pour diverses raisons

(métro, forêt, zone peu couverte par des antennes, etc..) alors j'ai pensé le système d'envoi des données par file d'attente. Lorsque la connexion est perdue, les données sont ajoutées à une liste d'envoi. Toutes les 5 minutes, on vérifie si la connexion est rétablie. Si oui, on envoie la file d'envoi.

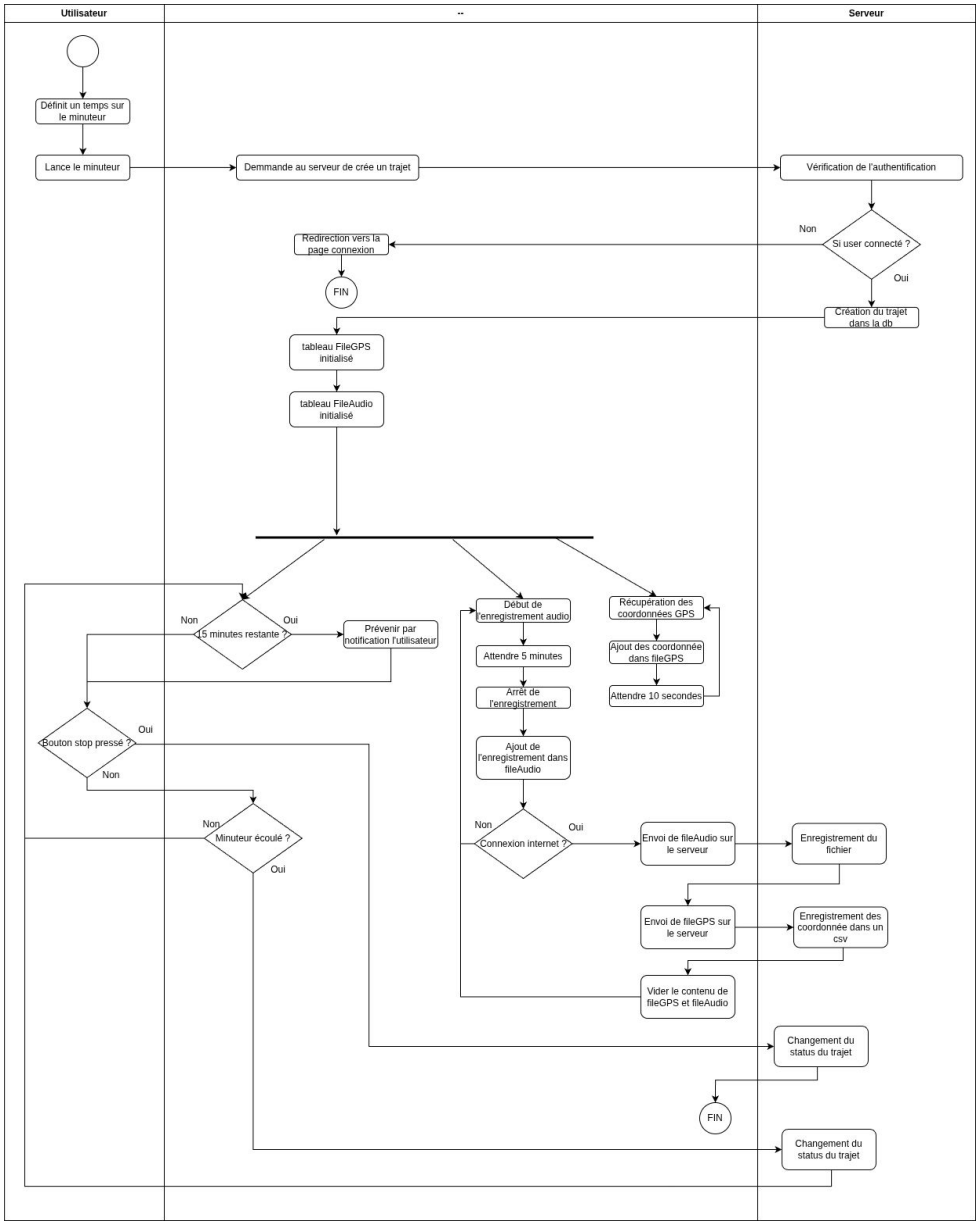


Diagramme d'activité de la logique métier du minuteur

Spécification Technique

Pour réaliser ce projet, l'application devait se faire avec une architecture N-tiers. Il s'agit d'une architecture séparant tous les niveaux de l'application (trois couches).

L’application est exécutée par plusieurs composants distincts (séparation par couche de responsabilités).
Une couche présentation, une couche applicative (service) et la couche d’accès aux données. Ces deux dernières couches contrairement à la première se trouvent toutes les deux du côté serveur. Toutes les parties de cette architecture sont capables de communiquer entre elles et peuvent donc coopérer en étant implantées sur des machines distinctes. Cela donne la possibilité de changer de base de données relationnelle sans impacter la couche de présentation ou la couche métier par exemple.

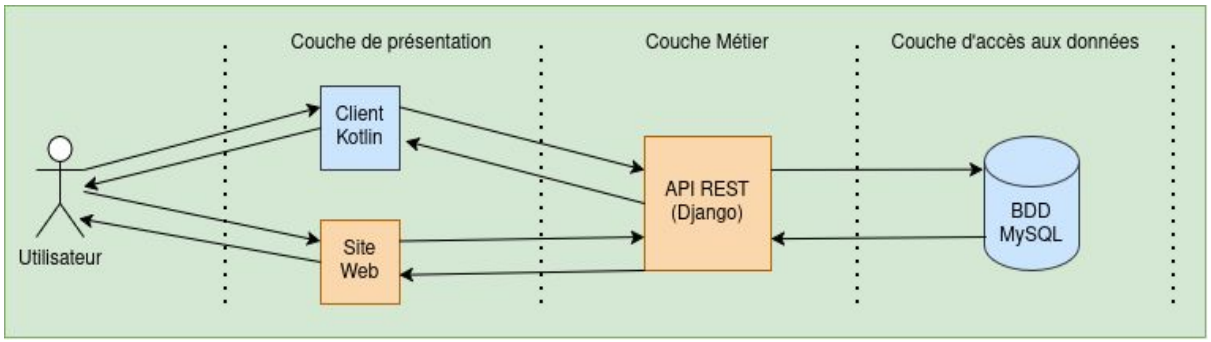


Figure 5: Schéma de l'architecture du projet

Couche de présentation

La couche de présentation, c’est le résultat final que voit l’utilisateur, on l'appelle l’Interface utilisateur. L’utilisateur va pouvoir interagir dessus et naviguer dans les différents menus et onglets. Elle permet aussi d’envoyer des données à la couche métier.
Pour ce projet, je devais avoir accès à la localisation, ainsi qu’au micro. J’aurais pu le faire avec une application mobile, mais en termes de simplicité pour l’utilisateur et pour ne pas être restreint par l'économiseur de batterie, j’ai décidé de faire une application téléchargeable.
Parmi les solutions de développement d’application existantes, j’ai décidé de me consacrer sur Kotlin plutôt que sur une solution cross-platform car le développement natif offre plus de performance qu’une solution cross-plateforme. De plus, Kotlin est créé et maintenu par Google, qui possède Android. Grâce à Kotlin, et l’utilisation de Jetpack Compose, je pouvais donc réaliser l’interface de manière simple, modulable et être compatible avec 99,2% des téléphones Android.

Couche métier

Cette partie, appelée également couche applicative se situe au cœur de l’application. C’est ici que les données, envoyées par la couche de présentation, sont analysées et

traitées. La couche métier fait le lien entre la couche de présentation et la couche d'accès aux données. Les actions, les vérifications et tout ce que l'utilisateur ne voit pas se font ici, souvent sur un serveur. On appelle ça une Interface de Programmation Applicative (API). Dans mon cas, l'architecture de l'API est de type REST (RESTful). Cela signifie que toutes les requêtes sont transmises via le protocole HTTP. Les données reçues et envoyées doivent être en JSON.
Pour réaliser mon projet, j'ai choisi le framework Python Django. En effet les spécificités de mon projet ne justifient pas l'utilisation d'une API comme Laravel car trop lourde. J'ai donc choisi Django pour sa légèreté et ses performances.

Couche d'accès aux donnée

Comme évoqué plus haut, la couche métier traite des données et doit les enregistrer quelque part. Pour ce faire, il y a la couche d'accès aux données. Il peut s'agir d'une base relationnelle ou d'un serveur de base de données NoSQL. Pour mon projet, j'ai choisi MySQL.

Les outils et technologies utilisés

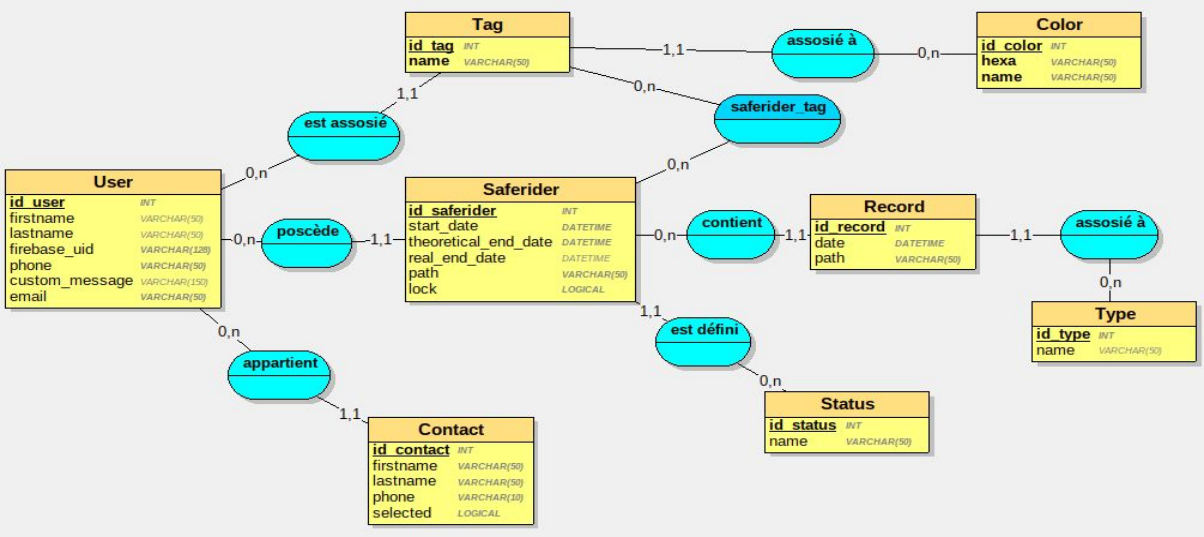
Pour réaliser ce projet, j'ai utilisé Android Studio pour le développement Kotlin. Android Studio est adapté aux développements Android grâce à son émulateur de téléphone intégré, ainsi que la compilation et le déploiement via un seul bouton, sur un téléphone émulé ou un un téléphone connecté via Wi-fi.
Pour ce qui est de l'API, Django étant un framework Python, j'ai utilisé PyCharm, l'outil créé par Jetbrains.

Modélisation conceptuelle de la base de données

La Modélisation conceptuelle de la base de données (MCD) est une représentation graphique de l'architecture des tables et des données que mon application va utiliser. J'ai donc réalisé plusieurs schémas

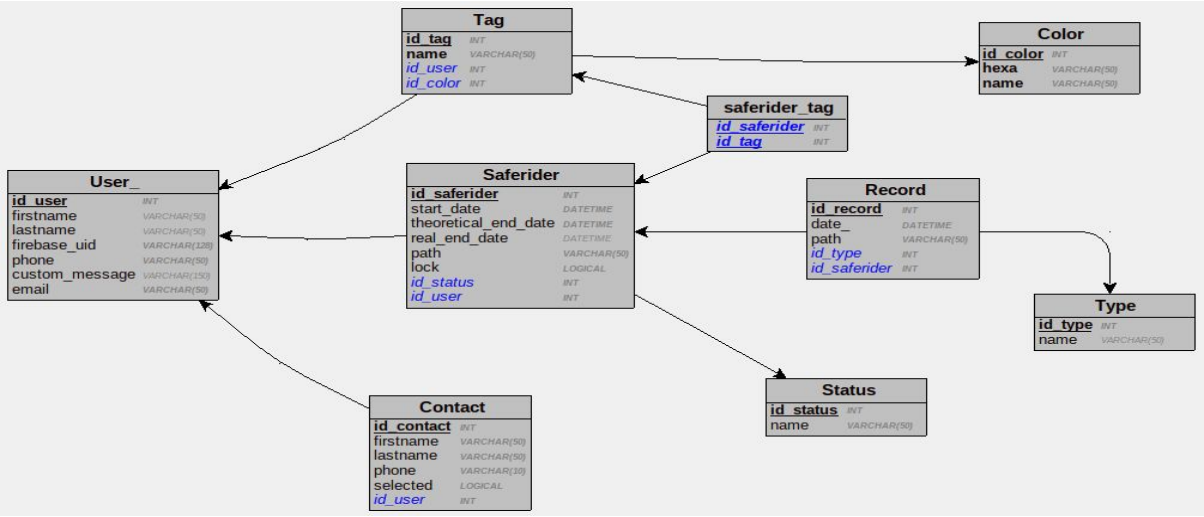
MCD

Le MCD est le schéma de départ, il permet de définir les champs dans les tables et les relations qu'ils vont avoir entre eux. Ces relations, on les appelle des cardinalités.



MLD

Le MLD (Modèle Logique des Données), est le dernier schéma, il représente la version finale et la plus représentative de la base de données. Les relations sont prises en compte et les tables pivot sont ajoutées.



Réalisation du projet

Pour la réalisation de mon projet, j'ai commencé par l'API.

Mise en place de l'API

Comme mentionné plus haut, j'ai choisi d'utiliser Django pour l'API de mon projet. Django nécessite un environnement Python. C'est grâce à celui-là que l'on peut lancer le projet. Étant donné que je travaille sur Ubuntu, Python est déjà installé par défaut. Il ne me reste plus qu'à créer le projet.

Installation de Django

Je me suis rendu dans mes Documents, et j'ai créé un dossier de projet. Puis j'ai défini un environnement virtuel Python. Cela me permet d'installer des dépendances qui ne serviront que dans ce projet. Pour ce faire, j'ai tapé la commande :

```
python3 -m venv env
```

Une fois l'environnement virtuel créé, il est nécessaire de l'activer, j'ai donc tapé cette commande :

```
source env/bin/activate
```

L'environnement est maintenant configuré. Je peux donc installer les dépendances nécessaires à mon projet. La première de toute, c'est le framework Django. Je l'installe avec pip, qui est le gestionnaire de paquets officiel pour Python.

```
pip install django
```

Ensuite j'ai créé le projet via cette commande :

```
django-admin startproject sentinelle
```

Le projet Django est maintenant installé. Il se compose de cette façon :

```
├── manage.py
└── sentinelle
    ├── asgi.py
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Le fichier principal est manage.py. C'est lui qu'on va exécuter pour lancer ou éteindre le projet. Ensuite il y a un dossier qui se nomme "sentinelle". Il contient les paramètres du projet (settings.py), les routes (urls.py) et deux fichiers de configuration servant à déployer le projet.

Pour commencer à travailler, il nous faut d'abord créer une application dans laquelle

il y aura tous nos composants métier de l'API. Pour ce faire, tapons la commande suivante :

```
python3 manage.py startapp api
```

On remarque qu'un dossier vient de se créer à la racine du projet.

```
├── api
│   ├── admin.py
│   ├── apps.py
│   ├── __init__.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── sentinelle
    ├── asgi.py
    ├── ...
```

Plusieurs fichiers sont contenus dans de dossier, et tous ont une utilité.

Fichier	Utilité
migrations/*	C'est dans ce dossier qu'on va stocker toutes les migrations de cette application. On les exécute via la commande : `python3 manage.py migrate`
models.py	Permet de définir la structure de la base de données. Elle présente les données et permet d'interagir avec elles grâce à son ORM.
admin.py	Ici on ajouter chaque model et grâce à ça, Django va créer automatiquement un back-office via les champs renseignés.
views.py	Ce fichier contient toutes nos fonctions utiles aux projets. Cela correspond à un contrôleur.

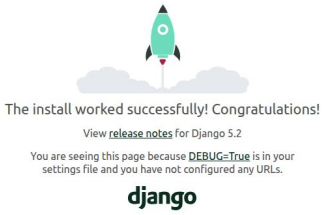
Une fois ces fichiers remplis, il m'est nécessaire d'appliquer les migrations avant de lancer le projet. Pour cela :

```
python3 manage.py migrate
```

puis :

```
python3 manage.py runserver
```

Nous pouvons voir que le projet est lancer en se rendant à cette l'url suivante : http://127.0.0.1:8000/



Mise en place de la couche d'accès aux données

Maintenant que notre projet est créé et fonctionne, il nous faut configurer notre base de données. Nativement, Django crée un fichier database.sqlite lorsque l'on lance les migrations.Cependant cette méthode n'est pas viable car ce n'est pas sécurisé et pas adapté pour un projet de cette taille. C'est pourquoi j'ai décidé d'utiliser MySQL.

Pour utiliser MySQL sur Django, il faut procéder différemment des autres frameworks. En effet, sur Laravel et Symphony, il est nécessaire d'installer le service MySQL. Cependant, avec Django, il faut installer le module MySQL via pip.

```
pip install mysqlclient
```

Puis, dans le dossier sentinelle/settings.py, il faut ajouter la déclaration de notre base de données :

```
DATABASES = {
       'default': {
           'ENGINE': 'django.db.backends.mysql',
           'NAME': os.environ.get('db_database'),
           'USER': os.environ.get('db_user'),
           'PASSWORD': os.environ.get('db_password'),
           'HOST': 'localhost',
           'PORT': '3306',
           'OPTIONS': {
               'charset': 'utf8mb4',
           }
       }
   }
```

Puis, accédons à MySQL en tapant "mysql" dans notre terminal, dans notre environnement virtuel, et tapons cette commande :

```
CREATE DATABASE sentinelle CHARACTER SET utf8mb4;
GRANT ALL PRIVILEGES ON sentinelle.* TO 'django'@'localhost';
FLUSH PRIVILEGES;
EXIT;
```

Et pour finir, définissons les valeurs d'accès dans un fichier .env à la racine du projet.

Il ne nous reste plus qu'à relancer les migrations, et notre base de données est fin prête !
Par la suite, j'ai donc ajouté toutes les tables et les champs dans des fichiers de migrations et de models. En voici un exemple complet.

```
class Migration(migrations.Migration):

    dependencies = [
        ('api', '0003_status_saferider'),
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name='SafeRiders',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True,
serialize=False, verbose_name='ID')),
                ('start_date', models.DateField()),
                ('theorotical_end_date', models.DateField()),
                ('real_end_date', models.DateField()),
                ('path', models.CharField(max_length=100)),
                ('locked', models.BooleanField(default=False)),
                ('id_status',
models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='api.status')),
                ('id_user',
models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
to=settings.AUTH_USER_MODEL)),
            ],
        ),
    ]
```

Fichier de migration de la table des trajets

```
class SafeRider(models.Model):
    start_date=models.CharField(max_length=100, null=True)
    theorotical_end_date=models.CharField(max_length=100, null=True)
    real_end_date=models.CharField(max_length=100, null=True)
    path=models.CharField(max_length=100, null=True)
    locked=models.BooleanField(default=False)
    tags = models.ManyToManyField("Tag", related_name="saferider", blank=True)
    id_status=models.ForeignKey(Status, on_delete=models.CASCADE)
    id_user=models.ForeignKey(CustomUser, on_delete=models.CASCADE)
    def __str__(self):
        return self.path
```

Fichier modèles de la table des trajets

```
from django.contrib import admin
from .models import *

admin.site.register(SafeRider)
```

Ajout du modèle dans le back-office dans le fichier [admin.py]

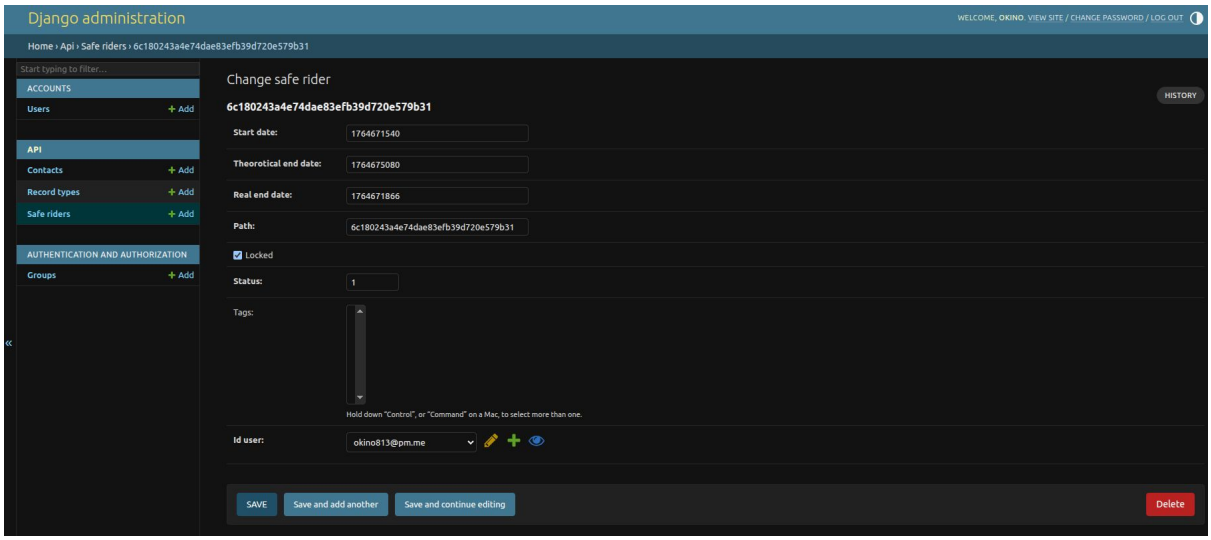


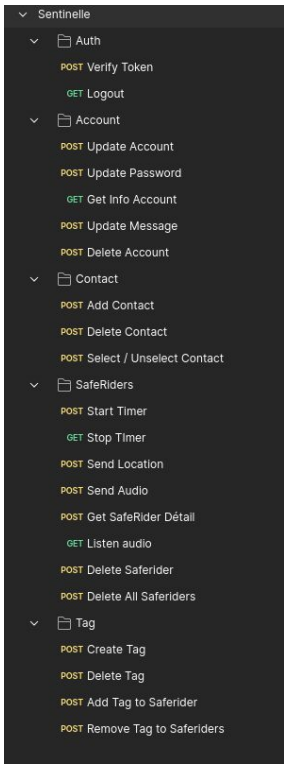
Figure 6: Exemple du back-office généré automatiquement par Django.

Ajout de l'API

Pour le fonctionnement de mon API, comme mentionné ci-dessus, j'ai créé une app réservée aux requêtes API. A l'intérieur, j'ai défini les routes dans le fichier urls.py

```
urlpatterns = [
   path('updateInfoAccount/', update_account,
name="update_account"),
   path('updatePassword/', update_password,
name="update_password"),
   path('updatemessage/', update_message,
name="update_message"),
   path('getinfoaccount/', get_account, name="get_account"),
   path('logout/', logout, name="logout"),
   path('addcontact/', add_contact, name="add_contact"),
   path('deletecontact/', delete_contact,
```

```
name="delete_contact"),
   path('updateselectcontact/', update_contact,
name="update_contact"),
   path('starttimer/', start_timer, name="start_timer"),
   path('stoptimer/', stop_timer, name="stop_timer"),
   path('sendlocation/', send_location,
name="send_location"),
   path('sendaudio/', send_audiofile, name="send_audiofile"),
   path('getsaferiderdetail/', get_saferider,
name="get_saferider"),

path('listen/saferiders/<str:saferider_path>/<str:filename>/'
, listen_audio, name="listen_audio"),
   path('deletesaferider/', delete_saferider,
name="delete_saferider"),
   path('createtag/', create_tag, name="create_tag"),
   path('deletetag/', delete_tag, name="delete_tag"),
   path('removetagfromsaferider/', unselect_tag,
name="unselect_tag"),
   path('addtagfromsaferider/', select_tag,
name="select_tag"),
   path('deleteAllSaferider/', delete_all_saferider,
name="delete_all_saferider"),
   path('deleteAccount/', delete_account,
name="delete_account"),
]
```



Pour chaque route, je les ai testées sur PostMan. PostMan, est un logiciel servant à tester les APIs, les automatiser et ainsi vérifier le bon fonctionnement de chaque route.

En effet sur PostMan, on peut créer des collections, et organiser nos requêtes dans des dossiers.

De mon côté, je m'en suis servi tout au long du développement de mon projet. En revanche, je n'ai pas automatisé les tests depuis PostMan car je souhaitais le faire directement dans l'API afin de ne pas dépendre de logiciel externe, et de pouvoir lancer les tests via un terminal.

L’authentification

Pour utiliser Sentinelle, il est nécessaire de pouvoir se connecter. Étant donné que le projet possède une API et une vue séparée de la logique métier, je dois mettre en place un système de token que je dois envoyer dans l’en-tête de chaque requête.

Prenons l’exemple d’une requête API, celle de la récupération des informations personnelles. Cette requête renvoie vers une fonction. Pour éviter de mettre la vérification du token dans chaque fonction, j’ai mis en place un décorateur.

Un décorateur permet d’étendre une fonction, d'utiliser le même code à plusieurs endroits, en n’ayant qu'à l'écrire qu’une seule fois. On le définit dans le fichier decorators.py. Puis on l’utilise juste au-dessus de la déclaration de la fonction, avec un @ devant.

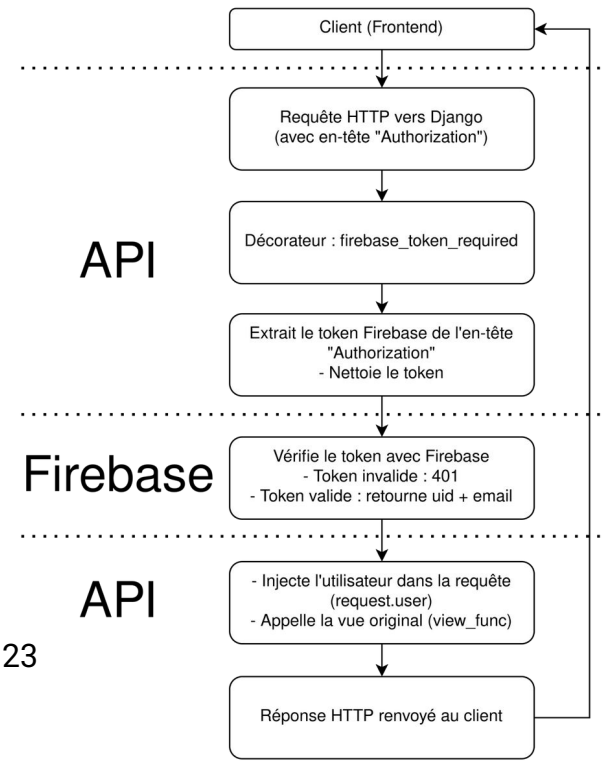
```
@firebase_token_required
@csrf_exempt
def get_account(request):
    user = request.user
    # On récupère les contacts de l'utilisateur
    contacts = user.contact_set.all()
    contact_list = []
    for contact in contacts:
        contact_list.append({
            'id': contact.id,
            'name': contact.name,
            'phone': contact.phone,
            'selected': contact.selected
        })

        …
```

Puis ensuite, la déclaration de la fonction décoratrice dans decorators.py :

```
def firebase_token_required(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):

        # Récupère l'en-tête Authorization
        auth_header = request.headers.get('Authorization')
        if not auth_header:
            return JsonResponse({'error': 'Token Firebase
manquant ou invalide'}, status=401)

        id_token = auth_header.split(' ')[1]  # Récupère juste
```

```
le token
        id_token = id_token.split(',')
        id_token = id_token[0]

        decoded_token =
firebase_auth.verify_id_token(id_token)
        try:
            uid = decoded_token['uid']
            email = decoded_token.get('email')

            user, created = CustomUser.objects.get_or_create(
                firebase_uid=uid,
                defaults={'email': email, 'username': email,
"phone": "", "message": ""}
            )
            request.user = user  # Injecte l'utilisateur

            return view_func(request, *args, **kwargs)

        except Exception as e:
            return JsonResponse({'error': str(e)}, status=401)

    return wrapper
```

Pour la gestion de l'authentification, j'ai décidé d'utiliser le service Firebase. Firebase est un service Google qui propose des outils pour gérer des applications web. J'ai décidé de m'en servir pour l'authentification. En effet, au lieu de gérer les tokens via mon API, en réalité je le fais sur Firebase.



Grâce à Firebase, j'ai mis en place une connexion via email et mot de passe, ainsi qu'un provider pour se connecter avec Google. L'authentification ne se fait donc pas de la même façon qu'un token JWT. En effet pour effectuer une requête, au lieu de vérifier le token sur le serveur, l'API va demander à Firebase si le token est valide.

De cette façon, je délègue la création et la connexion de compte à Firebase, et je ne stocke aucun mot de passe dans ma base de données. Tout se fait par

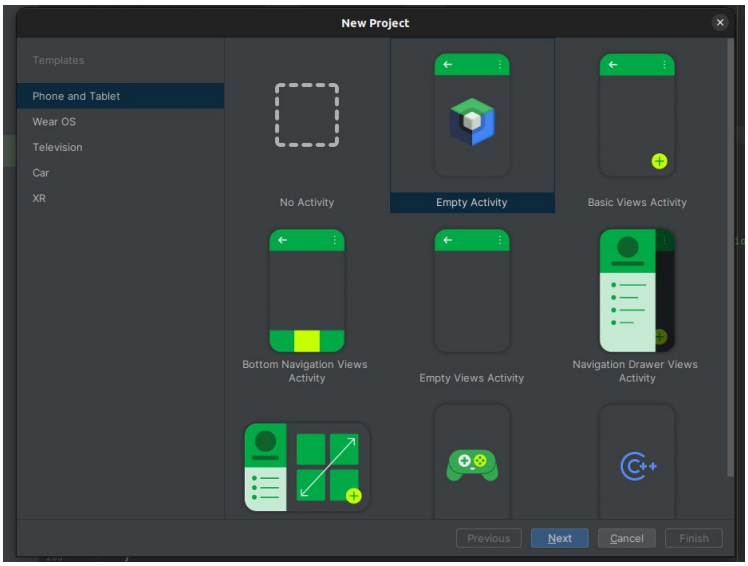
l’email et le token que je sauvegarde.

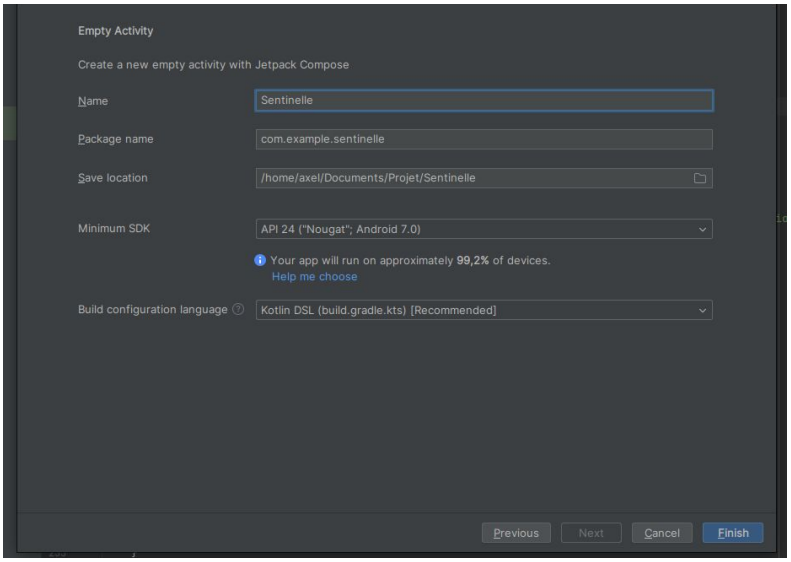
Comme le démontre le schéma, le client (Kotlin) envoie le jeton JWT à l’API. L'API fait des traitements dessus et le transmet à Firebase. Firebase le déchiffre avec sa clé privée et renvoie le succès ou l’échec.

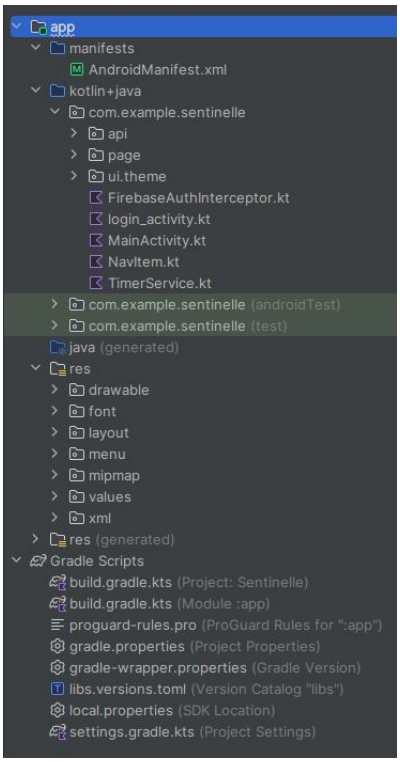
L’avantage est qu’en cas de piratage, les mots de passe ne seront pas exposés.

Couche de présentation

Après avoir installé Android Studio et le JDK souhaité, j’ai créé un nouveau projet en choisissant d’utiliser Jetpack Compose.







Mon projet se crée automatiquement et Jetpack Compose est déjà configuré. La structure d’un projet Kotlin se compose de cette manière :

Le projet est contenu dans un dossier qui porte le nom du projet, dans mon cas c’est Sentinelle. A l’intérieur nous avons un dossier app et Gradle Scripts.

Le Gradle Scripts contient tous les paramètres essentiels à la compilation du code. Nous avons la version du SDK, de l’API Android, le namespace de l’application, la liste des dépendances associées et plein d'autres paramètres qui ne nous intéressent pas spécialement ici.

Le dossier app, en revanche, nous intéresse beaucoup car

c’est dedans que le code du projet se trouve. Nous avons 3 dossiers en excluant les dossiers générés automatiquement. Les dossiers qui nous intéresses sont donc :

- manifests, contient un seul fichier, le AndroidManifest.xml
- kotlin+java, contient les fichiers que nous ajouterons pour le projet
- res, contient toutes les ressources, médias, composants XML, valeurs

Regardons en détail ce qu’il se trouve dans kotlin+java. Il y a 3 dossiers similaires mais pour des utilisations différentes. Le premier contient le code que nous ajouterons pour développer l’application. Les deux autres servent à effectuer des tests unitaires et fonctionnels. Nous reviendrons dessus plus tard.

Interface utilisateur avec Jetpack Compose

Jetpack Compose est un framework basé sur Kotlin qui permet de créer et gérer des interfaces utilisateur avec du code, contrairement à avant où il fallait définir l’interface dans des fichiers XML. Cela permet de rendre le code modulaire, réutilisable et de rafraîchir des interfaces utilisateur avec des données.

```
@Composable
fun TutorialScreens(colors : List<Color>, onTutorialFinished: () -> Unit) {
    var index by remember { mutableStateOf( value = 0) }

    when (index) {
        0 -> TutoPage(title = "Un minuteur pas comme les autres ...", text = "
            imageRes = R.drawable.logo_minuteur,
            color = colors[0],
            fleche = R.drawable.fleche_tuto1) {
            index++
        }
        1 -> TutoPage(title = "Suivie de votre parcours", text = "Votre trajet
            imageRes = R.drawable.logo_carte_tuto2,
            color = colors[3],
            fleche = R.drawable.fleche_tuto2) {
            index++
        }
        2 -> TutoPage(title = "Environnement sonore", text = "Votre environnem
            imageRes = R.drawable.logo_record_tuto3,
            color = colors[1],
            fleche = R.drawable.fleche_tuto3) {
            index++
        }
        3 -> TutoPage(title = "Prévenir vos proches", text = "Ajoutez des cont
            imageRes = R.drawable.logo_proche_tuto_4,
            color = colors[4],
            fleche = R.drawable.fleche_tuto4) {
            onTutorialFinished()
        }
    }
}
```

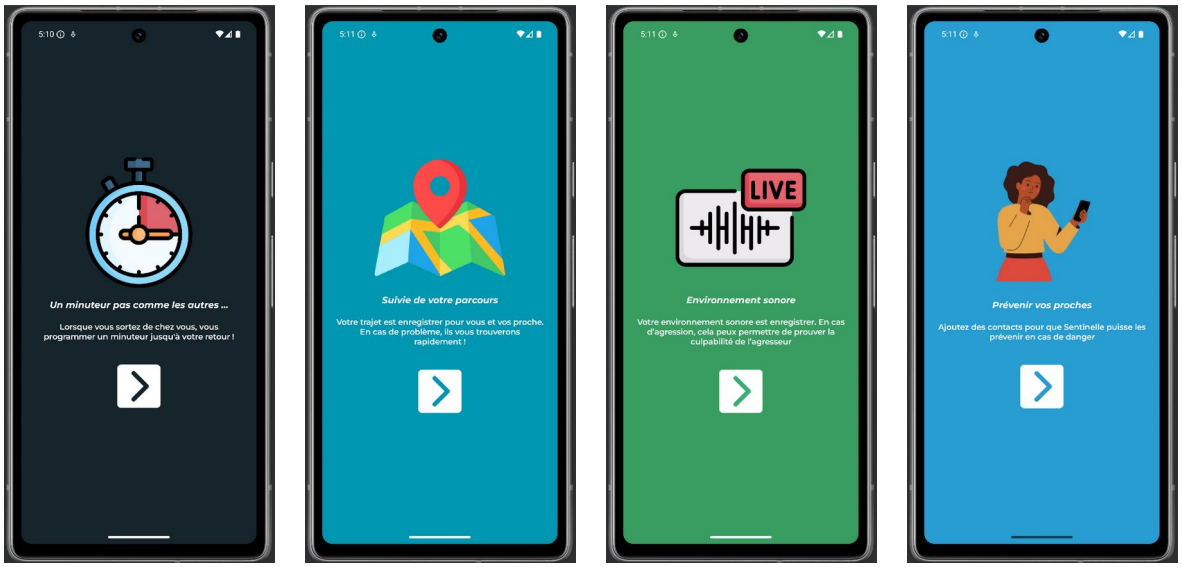
C’est donc avec ce framework que j’ai conçu l’interface utilisateur de mon application. Voici un exemple de code modulaire réutilisable que j’ai mis au point avec Jetpack Compose.

Il s’agit de l’affichage des pages tuto lorsque l’utilisateur n’est pas connecté. TutorialScreens gère la construction des pages de tuto de façon dynamique. Vu que les pages tutos sont toutes construites pareil, j’ai fait de la page, un bloc défini dans TutoPage.

Le bloc TutoPage récupère en argument le titre, l’image, la couleur et la fonction qui passe à la page suivante lors du clique sur le bouton.

```kotlin
@Composable
fun TutoPage(title: String, text: String, imageRes: Int, color: Color, fleche: Int, onNext: () -> Unit) {
    UpdateStatusBarColor(color, context = LocalContext.current)
    Surface(modifier = Modifier.fillMaxSize(), color = color) {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding( all = 20.dp),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            Image(painter = painterResource(id = imageRes), contentDescription = null, modifier = Modifier.align(Alignment.CenterHorizontally))
            Spacer(modifier = Modifier.padding( all = 10.dp))
            Text(
                text = title,
                color = Color.White,
                fontSize = 16.sp,
                textAlign = TextAlign.Center,
                fontFamily = Montserrat,
                fontWeight = FontWeight.Bold,
                fontStyle = FontStyle.Italic,
            )
            Spacer(modifier = Modifier.padding( all = 10.dp))
            Text(
                text = text,
                color = Color.White,
                fontSize = 14.sp,
                textAlign = TextAlign.Center,
                fontFamily = Montserrat,
                fontWeight = FontWeight.SemiBold,
                fontStyle = FontStyle.Normal,
            )
            Spacer(modifier = Modifier.padding( all = 20.dp))
```

Ainsi, on obtient les pages suivantes :



Permission du téléphone

Durant le parcours de ces pages, les permissions du téléphone sont demandées. Nous avons les permissions suivantes :

- Position exacte
- Micro
- Notification
- Ignorer les optimisations de l'économiseur de batterie

Ce sont les permissions indispensables pour le bon fonctionnement de l'application.

Certaines permissions sont demandées à l'utilisateur, comme celle citées ci-dessus, et d'autres n'ont pas besoin d'approbation, mais elles doivent tout de même être déclarées.

Pour ce faire, rendons nous dans le dossier *app/manifests*. A l'intérieur du fichier AndroidManifest.xml, nous avons ceci :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <!-- Permissions nécessaires -->
    <uses-permission
android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE" />
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE_DATA_SYNC
" />
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE_MEDIA_PLA
YBACK" />
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE_LOCATION"
/>
    <uses-permission
android:name="android.permission.POST_NOTIFICATIONS" /> <!--
Permission pour enregistrer l'audio -->
    <uses-permission
android:name="android.permission.RECORD_AUDIO" />
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission
android:name="android.permission.ACCESS_BACKGROUND_LOCATION"
/>
    <uses-permission
android:name="android.permission.WAKE_LOCK" />
```

```
    <!-- Pour Android 9+ (API 28+) : éviter les optimisations
de batterie -->
    <uses-permission
android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIM
IZATIONS" />

    <!-- Pour Android 12+ (API 31+) : service en arrière-plan
-->
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE_MICROPHON
E" />
```

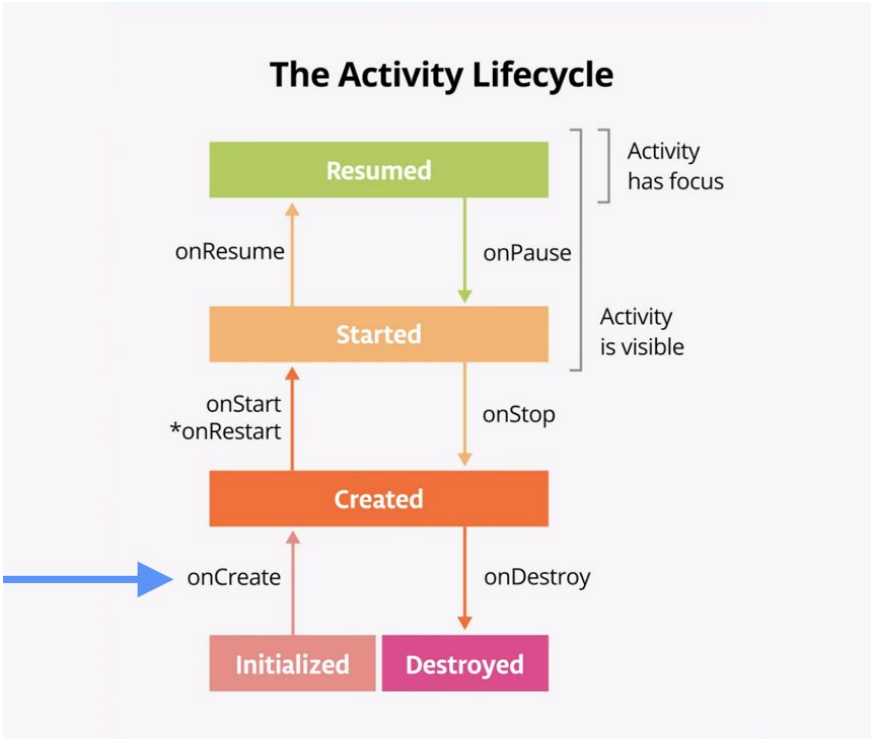
Comme vous pouvez le constater, les permissions sont déclarées par les tags `uses-permission`. C'est dans ce fichier que l'on définit également les activités et les services de l'application.

Une activité est une classe, un composant d'interface utilisateur qui représente un écran avec lequel l'utilisateur peut interagir. L'activité gère le cycle de vie de l'écran (création, pause, reprise, destruction …) et possède un état qui lui est propre, lui permettant alors de créer un flux de navigation.

Pour fonctionner, une activité doit être déclarée dans le manifest :

```
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:screenOrientation="portrait"
    android:theme="@style/Theme.Sentinelle.Splash">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category
android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

</activity>
```

L'activité "MainActivity" est maintenant déclarée, on peut la retrouver dans le dossier app/kotin+java/com.example.sentinelle. Une activity se compose de cette façon :



Tout commence par la fonction onCreate. Elle s'exécute lorsque l'activité s'ouvre pour la première fois. C'est elle qui va construire l'interface et la renvoyer à l'utilisateur. C'est dans cette fonction que je teste si l'utilisateur est connecté ou non.

La Navigation de composant

Si l'utilisateur est connecté, alors on va afficher l'interface principale. Pour rendre cet interface fluide et modulable, j'ai fait un composant principal, qui appelle tout les autres. Ce composant, c'est le BottomMenu.

En effet pour fonctionner, je n'exécute qu'une seule fonction composable qui appelle toutes celles dont j'ai besoin. Ma fonction BottomMenu récupère les couleurs et les données que j'ai besoin pour construire le reste. On appelle ça une fonction Stateful. Cette fonction récupère les données qui peuvent être modifié plus tard, et les transmet à la fonction composable stateless. Un composant stateless, ne servira que pour construire un élément / bloc. Elle ne gère aucune donnée et ne réalise aucun processus.

En voici un exemple simplifié :

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        // Défini l'interface à crée et à afficher
        setContent {
            // Charge les variable de theme
```

```
        MyApplicationTheme {
            Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                // Appelle la fonction composable Stateful
                Ecran1(
                    context = LocalContext.current,
                    modifier = Modifier.padding(innerPadding)
                )
            }
        }
    }
}

// Composant Stateful
@Composable
fun Ecran1(
    context : Context,
    modifier: Modifier
){
    // Défini la variable
    var firstname by remember {mutableStateOf("")}

    // Envoi la variable dans le composant stateless
    Formulaire(
        username = firstname,
        onChangeName = {firstname = it}
        )
}

// Composant Stateless
@Composable
fun Formulaire(
    username: String,
    onChangeName: (NewUserName: String) -> Unit
){
    Box(){
        Column(
            modifier = Modifier
                .padding(16.dp)
                .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center,
        ){
```

```
        Text("Bonjour $username")
        Text("Entrez votre nom :")
        TextField(
            modifier = Modifier.fillMaxWidth(),
            value = username,
            onValueChange = {
                onChangeName(it)
            },
            placeholder = { Text("Nom d'utilisateur") }
        )
    }
  }
}
```

Nous pouvons voir dans ce code, que la variable du nom est définie dans la fonction composable Ecran1. Cette fonction appelle le Formulaire en lui donnant le contenu de cette variable. Lorsque l'utilisateur modifie le champ, le nom remonte vers la fonction racine (Ecran) et la valeur est modifiée dans cette dernière.

Si la valeur du name avait été modifiée dans la fonction Formulaire, l'interface ne se serait pas actualisé. Cette façon de gérer les données et de les afficher fait partie des bonnes pratiques de Kotlin. Elle permet de ne recomposer que les composants qui utilise cette donnée, ainsi on optimise les performances en évitant de tout recomposer juste pour qu'une valeur soit changée.

Pour le cas de mon BottomMenu, c'est de cette façon que ça fonctionne. Ma fonction de navigation génère une variable, un index :

```
var selectedIndex by remember {
    mutableIntStateOf(0)
}
```

Ensuite elle affiche une barre de navigation où elle attribue à des boutons de redirection, un index. Puis elle appelle une fonction composable avec en argument, l'index actuel. C'est cette dernière fonction qui va afficher les composants lorsqu'on aura cliqué dessus.

```
when(selectedIndex){
    0-> HomeScreen(
        colorList,
        modifier = modifier
    )
    1-> AppNavigation(
        colorList,
        AppValues.saferiders,
    )
```

```
    2-> NavigationTabExample(
        colorList,
        modifier = modifier,
    )
    3-> SettingsScreen(
        modifier = modifier,
        context = context,
        colors = colorList,
        sharedPreferences = sharedPreferences,
        isLoggedIn = isLoggedIn,
        isContrast = isContrast,
        onChangeColor = onChangeColor,
        googleSignInClient = GoogleSignIn.getClient(
            context,

GoogleSignInOptions.Builder(GoogleSignInOptions.DEFAULT_SIGN_IN)
                .requestIdToken(context.getString(R.string.web_client_id))
                .requestEmail()
                .build()

    )
  }
```

Vous l'aurez peut être remarqué, mais les boutons du menu se trouvent dans le composable Stateful. Je l'ai mis à l'intérieur car les boutons ne sont pas susceptibles de changer. Ils sont immuable. Donc je peux l'afficher directement dans cette dernière. Contrairement aux composants d'interface qui eux changent selon le bouton cliqué.

La connexion au compte utilisateur

Étant donné que je conçois une application mobile, je ne peux pas gérer la connexion comme une page web. Dans une application mobile, on stocke ce qu'on appelle un refresh token. Ce refresh token, va générer un token avec une durée de vie d'environ une heure. C'est ce token qui est envoyé à l'API.

Le refresh token, est générer par le SDK lors de la connexion utilisateur. Avec Firebase, elle fonctionne comme ceci :

1) Récupération des credentials (Email, Mot de passe)
2) Vérification des informations depuis le SDK
3) Attribution d'un refresh token

Côté code, ça ressemble à ça :

```
fun signUpWithEmail(email: String, password: String, callback: (Boolean) -
```

```
> Unit) {
    auth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener { task ->
        if (task.isSuccessful) {
            val user = auth.currentUser
            Toast.makeText(this, "Compte créé !",
Toast.LENGTH_SHORT).show()
            // Mettre à jour l'état de connexion
            isLoggedIn.value = true
            sharedPreferences.edit().putBoolean("is_authentificated",
true).apply()
            callback(true)
        } else {
            Toast.makeText(this, "Erreur: ${task.exception?.message}",
Toast.LENGTH_SHORT).show()
            callback(false)
        }
    }
}
```

Cette fonction est une fonction que j'ai créée et qui en appelle une autre, celle de Firebase qui crée un compte utilisateur via l'adresse mail. Lorsque Firebase me valide la création du compte, l'utilisateur est automatiquement connecté. C'est pourquoi je défini dans le sharedPreferences, la variable « is_authentificated ». Lors du lancement de l'application, cette variable va être tester et déterminer sur quelle page rediriger l'utilisateur.

Les requêtes

Comme vu plus haut, pour déterminé qui envoi une requête à l'API, un token est inséré dans le header. Dans le cas de mon client Kotlin, ce token est renvoyer par le SDK Firebase.

Pour chaque requêtes, un Interceptor récupère le token et l'injecte automatiquement dans la requête.

```
override fun intercept(chain: Interceptor.Chain): Response {
    val originalRequest = chain.request()
    val user = FirebaseAuth.getInstance().currentUser

    return if (user != null) {
        try {
            // Obtenir un token valide
            val tokenTask = user.getIdToken(true)
            val tokenResult = Tasks.await(tokenTask)
            val token = tokenResult.token ?: throw Exception("Token Firebase
```

```
invalide")

        // Ajouter le token à la requête
        var requestWithToken = originalRequest.newBuilder()
            .addHeader("Authorization", "Bearer $token")
            .build()

        var response = chain.proceed(requestWithToken)

        // Rafraîchir le token si besoin
        if (response.code == 401) {
            response.close()
            val refreshedToken = Tasks.await(user.getIdToken(true)).token
                ?: throw Exception("Impossible de rafraîchir le token
Firebase")

            requestWithToken = originalRequest.newBuilder()
                .header("Authorization", "Bearer $refreshedToken")
                .build()

            response = chain.proceed(requestWithToken)
        }
```

Pour chaque requête faite à l'API, un token sera automatiquement injecter dans l'en-tête de la requête. Cela permet de réduire la longueur des fonctions d'appel. Cependant je trouvais le longueur encore trop longue. C'est pourquoi j'ai créé deux fonctions principal : « apiGet » et apiPost ». J'appelle ces fonctions avec un endpoint en argument. Cela permet de réduire 22 lignes par fonction d'appel à l'API.

```
fun apiPost(
    context: Context,
    endpoint: String,
    json: JSONObject,
    onSuccess: (JSONObject) -> Unit,
    onError: () -> Unit
) {
    val body =
json.toString().toRequestBody("application/json".toMediaType())
    val request = Request.Builder()
        .url("${AppValues.base_url}/api/$endpoint/")
        .post(body)
        .build()
```

```
    buildClient().newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) = onError()

        override fun onResponse(call: Call, response: Response) {
            val bodys = response.body?.string()
            if (response.isSuccessful && bodys != null) {
                try {
                    onSuccess(JSONObject(bodys))
                } catch (e: JSONException) {
                    onError()
                }
            } else {
                onError()
            }
        }
    })
}
```

L'envoi des données

Une des questions qui m'a demandé beaucoup de réflexion, est la transmission des données audio pendant la durée d'écoulement du minuteur.

Solution	Avantage	Inconvénient
Envoi direct	Pas de stockage sur le téléphone	Perte de donnée si pas de connexion
		Consomme beaucoup de donnée mobile et de batterie
		Demande des ressources serveur important
Envoi toute les x minutes	Beaucoup moins de requêtes sur le serveur	Si pas de réseaux, pas d'envoi
		Attente de x minutes pour le proche en cas de problème
Envoi toute les x minutes et stockage dans une file d'attente	Envoi de la file d'attente lorsque la connexion est retrouvé, sans perte de donnée	Attente de x minutes pour le proche en cas de problème

J’ai fini par choisir la solution 3, et de fixer un cycle d’enregistrement à 5 minutes. Le temps par cycle peut être vu comme un détail insignifiant mais en réalité il est important car il va influencer l’expérience utilisateur.

Si on met un délai trop court, par exemple 2 minutes, lorsque l’utilisateur va afficher la liste des ses enregistrements, il va y en avoir beaucoup, surtout si la durée du trajet est de plusieurs heures.

A l’inverse, si on configure un délai trop long, par exemple 7 minutes, dans le cas où l’utilisateur est en danger et qu’un proche consulte son trajet, il devra attendre 7 minutes afin d’obtenir un nouvel enregistrement sonore, ce qui peut augmenter le stress du proche. C’est pourquoi le délai de 5 minutes m’a semblé être la meilleure option.

Les services sur Kotlin

Comme vu plus tôt, un composable / une activité possède un état qui lui est propre. De ce fait, le minuteur peut se couper par lui même lorsque l’utilisateur change de fenêtre, change d’application en cours, voir même verrouille son téléphone.

Pour contrer cela, il existe sur Kotlin, les Services. Un Service est un composant Android qui permet de faire exécuter du code en arrière-plan, sans interface utilisateur. Pour qu’un service persiste même après avoir fermé l’application, il lui est obligatoire de générer une notification permanente. C’est cette notification qui donne vie au Service.

Contre le problème de changement de composable, d’application ou de verrouillage d’écran, le service est parfaitement adapté. Le Service possède lui aussi un cycle de vie.

onCreate()	Est appelé automatiquement à la création du Service
onStartCommand()	Est appelé à chaque appel de startService
onBind()	Est appelé quand un composant veut se connecter
onDestroy()	Appelé juste avant que le service soit détruit

Dans mon projet, j’ai créé un fichier dédié au service qui contiendra l’entièreté de la logique du service. Je l’ai nommé « TimerService ».

Le minuteur se lance à l’appelle de onStartCommand.

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    when (intent?.action) {
        "START_TIMER" -> {
            val totalSeconds = intent.getIntExtra("totalSeconds", 0)
            startForeground(1, buildNotification())
            startCountdown(totalSeconds)
        }
        "RESUME_TIMER" -> {
            val totalSeconds = intent.getIntExtra("totalSeconds", 0)
            startForeground(1, buildNotification())
            startCountup(totalSeconds)
        }
        "STOP_TIMER" -> {
            stopCountdown()
            stopLocationUpdates()
        }
    }
    return START_STICKY
}
```

Cette fonction détermine l'action à faire. Dans le cas du lancement, elle va créer une notification, et lancer le compte à rebours grâce à startCountDown.

Cette fonction va géré elle même les actions à faire dans le temps.

```
timer = object : CountDownTimer(totalSeconds * 1000L, 1_000L) {
    override fun onTick(millisUntilFinished: Long) {
        secondsElapsed++

        // Action chaque seconde
        var heureRestant = millisUntilFinished / 3600_000
        var minuteRestant = (millisUntilFinished % 3600_000) / 60_000
        var secondeRestant = (millisUntilFinished % 60_000) / 1000

        AppValues.hour.value = heureRestant.toInt()
        AppValues.minute.value = minuteRestant.toInt()
        AppValues.seconde.value = secondeRestant.toInt()

        Log.d("TimerService", "Heure: $heureRestant, Minute:
$minuteRestant, Seconde: $secondeRestant")

        if(secondsElapsed % 10 == 0) {
            // Action toutes les 10 secondes
            Log.d("TimerService", "Action toutes les 10 secondes")
            startLocationUpdates()
```

```
        }

        if(secondsElapsed % 300 == 0) {
            // Action toutes les 5 minutes
            handleAudioRecording()

            // Tenter d'envoyer les queues si connexion disponible
            if (isNetworkAvailable()) {
                sendQueuedAudioFiles()
                sendQueuedLocations()
            } else {
                Log.d("TimerService", "Pas de connexion - données en queue
(Audio: ${audioFileQueue.size}, GPS: ${locationQueue.size})")
            }
        }
    }
```

Comme vous pouvez le constater, un enregistrement audio est commencé. Toutes les 5 minutes, il s'arrête, s'enregistre dans un tableau et en commence un autre. C'est la fonction handleAudioRecording qui regroupe toutes les actions concernant l'audio.

Grâce à MediaRecorder, je peux enregistrer le microphone dans un fichier au format M4A. Je peux également définir le codec et d'autres informations concernant la qualité de l'audio

```
mediaRecorder = MediaRecorder().apply {
    setAudioSource(MediaRecorder.AudioSource.MIC)
    setOutputFormat(MediaRecorder.OutputFormat.MPEG_4)   // conteneur
MP4
    setAudioEncoder(MediaRecorder.AudioEncoder.AAC)      // codec AAC
    setAudioEncodingBitRate(128000)                      // encodage 128kbps
    setAudioSamplingRate(44100)                          // qualité CD
    setOutputFile(currentAudioFile?.absolutePath)

    prepare()
    start()
}
```

En parallèle de l'enregistrement audio, les coordonnées GPS sont récupérées.

Comme on peut le voir, la fonction startLocationUpdates est appelée toute les 10 secondes. Cette fonction va renvoyer un callback, le locationCallback.

```
val locationRequest =
```

```
com.google.android.gms.location.LocationRequest.Builder(
    10_000L // 10 secondes
)
    .setPriority(com.google.android.gms.location.Priority.PRIORITY_HIGH_AC
CURACY)
    .build()

fusedLocationClient.requestLocationUpdates(
    locationRequest,
    locationCallback,
    mainLooper
)
```

Ce CallBack, est défini dans le onCreate et se présente de cette manière :

```
locationCallback = object :
com.google.android.gms.location.LocationCallback() {
    override fun onLocationResult(result:
com.google.android.gms.location.LocationResult) {
        for (location in result.locations) {
            Log.d("TimerService", "Localisation → lat: ${location.latitude}, lng:
${location.longitude}")

            // Ajouter à la queue
            val locationData = LocationData(
                latitude = location.latitude.toString(),
                longitude = location.longitude.toString(),
                timestamp = System.currentTimeMillis() / 1000
            )
            locationQueue.add(locationData)
            Log.d("TimerService", "Coordonnées ajoutées à la queue ($
{locationQueue.size} en attente)")

            // Tenter d'envoyer immédiatement si connexion disponible
            if (isNetworkAvailable()) {
                sendQueuedLocations()
            }
        }
    }
}
```

Il va récupérer la mise à jour de la position et l'ajouter à la queue ou l'envoyer directement si la connexion est disponible.

Pour arrêter le minuteur, la fonction onStartCommand est appelée mais cette fois,

avec l'intent « STOP_TIMER ». Elle appellera donc deux fonctions :

- stopCountdown()
- stopLocationUpdate()

Cette dernière, supprimera le Callback créé dans le onCreate.

Validation du projet

Si nous regardons notre diagramme en V, nous pouvons remarquer que la phase ascendante du projet est la validation. Pour cela, j'ai effectué des tests. Cela permet de tester des fonctionnalités, des bouts de code et lorsque ces tests sont automatisés, de pouvoir, avec une seule commande, de tester son application à tout moment, souvent après un déploiement.

Les types de test

Ces tests qu'on fait subir à notre application, ont plusieurs nom pour plusieurs utilisations.
Ces tests sont appelé :

- Tests unitaires
- Tests fonctionnels
- Tests d'intégration

Le test fonctionnel, vérifie que l'application respecte les spécifications fonctionnelles. Ces tests se placent du point de vue de l'utilisateur, contrairement au test unitaire, où le développeur a accès au code interne. Le test permet de tester le parcours utilisateur, en utilisant l'interface graphique.

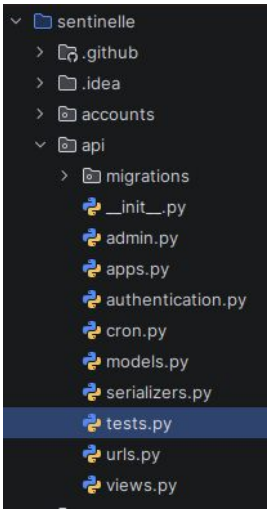
Test sur l'API

Le test unitaire, permet de vérifié le fonctionnement d'un bout de code isolé comme une fonction, une méthode ou une classe. Le but est de tester le comportement d'une fonction indépendamment du reste du code. Pour l'api, j'ai effectué les tests unitaires suivant :

Validation des données de contact	Tester la logique de validation lors de l'ajout d'un contact (format téléphone, nom obligatoire)
Génération du path unique SafeRider	Vérifier que la fonction de génération du path produit bien des identifiants uniques

	et conformes
Calcul de la date théorique de fin	Tester le calcul du timestamp de fin théorique à partir du timestamp de début + durée
Vérification des conditions de démarrage	Tester la logique qui vérifie si un utilisateur peut démarrer un SafeRider (contacts, message, téléphone, etc.)
Détermination du statut à l'arrêt	Tester la logique qui détermine si le statut doit être 1 ou 2 selon si l'arrêt est avant ou après la fin théorique
Validation du format des coordonnées GPS	Vérifier que les coordonnées latitude/longitude reçues sont dans un format valide
Logique de vérification d'accès aux fichiers	Tester la fonction has_access_to_path qui vérifie qu'un utilisateur a bien accès à un SafeRider

Au niveau du code, les tests doivent être écrits dans le fichier test.py situé dans le répertoire d'application.



A l'intérieur, je définis une classe, mes fonctions de test. A ces fonctions, nous en ajoutons une, la fonction setUp, qui permet de moquer les éléments nécessaires au fonctionnement des tests.

Un Mock, est un objet simulé qui reproduit le comportement d'un objet réel. Cela permet de simuler, par exemple, un token d'authentification pour pouvoir tester des fonctions comme l'ajout d'un contact où la connexion est obligatoire.

Reprenons ce même exemple, et regardons ce qu'il se passe au niveau du code.

```
def test_1_validation_contact_phone_format(self):
    # Format valide
    valid_phone = "0612345678"
    self.assertEqual(len(valid_phone), 10)
    self.assertTrue(valid_phone.isdigit())
    self.assertTrue(valid_phone.startswith('0'))

    # Format invalide
    invalid_phones = ["061234567", "06123456789", "abc1234567", "1612345678"]
    for phone in invalid_phones:
        self.assertFalse(
```

```
        len(phone) == 10 and phone.isdigit() and phone.startswith('0')
    )
```

Tout d'abord, on tente d'insérer un numéro de téléphone valide. Pour vérifier le résultat, on utilise la fonction assertEqual. Cette fonction vérifie si deux arguments renseignés sont identiques. De cette façon nous vérifions que la longueur totale du numéro de téléphone est bien égale à 10. Ensuite nous ajoutons plusieurs traitements tel que la vérification du type et du contenu. Lorsque ces vérification échoue, une erreur apparaît dans mon terminal. Si elle réussit, elle renvoie un succès.

Dans Django, pour lancer ces tests, je tape la commande python3 manage.py test. Et nous pouvons constater qu'aucune erreur n'est retournée :

```
(env) axel@lapsa:~/Documents/Projets/Sentinelle/api/sentinelle$ python3 manage.py test -v 0
/home/axel/Documents/Projets/Sentinelle/api/sentinelle/media
System check identified no issues (0 silenced).
id_contact 1
username func@test.com
/home/axel/Documents/Projets/Sentinelle/api/sentinelle/media/saferiders/60a9e02310ea4b56bb9bb0d8a332ae8e11771063031.143004/audio_1234567890.m4a
saferiders/60a9e02310ea4b56bb9bb0d8a332ae8e11771063031.143004/audio_1234567890.m4a
----------------------------------------------------------------------
Ran 21 tests in 3.718s

OK
```

Les 21 tests, comprennent également les tests d'intégration.

Le test d'intégration, vérifie que plusieurs composants (déjà testés individuellement) interagissent correctement. Le seul résultat compte à la fin du test et c'est celui là qui déterminera la réussite. En voici des exemples :

Authentification Firebase et création d'utilisateur	Tester le flux complet `verify_token` avec un vrai token Firebase et vérifier la création/récupération d'utilisateur
Écriture et lecture du fichier CSV de coordonnées	Envoyer plusieurs positions via send_location puis les récupérer via get_saferider pour vérifier la cohérence
Upload et téléchargement de fichiers audio	Uploader un fichier audio via `send_audiofile`, puis le récupérer via `get_audio` ou `listen_audio`
Interactions entre tags et SafeRiders	Créer des tags, les associer à plusieurs SafeRiders différents, vérifier la relation many-to-many
Suppression en cascade lors du delete_account	Supprimer un compte et vérifier que tous les SafeRiders, tags, contacts et fichiers associés sont supprimés
Gestion de la structure de fichiers	Vérifier que la création de dossiers SafeRider, les permissions, et

	l'organisation des fichiers (CSV + audios) fonctionnent correctement
Authentification et autorisation d'accès	Tester qu'un utilisateur A ne peut pas accéder aux SafeRiders/fichiers d'un utilisateur B (test de sécurité)

J’ai également fait des tests fonctionnels. Un test fonctionnel, sert à tester le fonctionnement complet d’un endpoint d’un API. On simule une requête HTTP et on vérifie sa réponse. En voici des exemples :

Cycle complet de gestion d'un contact	Créer, modifier la sélection, puis supprimer un contact en vérifiant le résultat à chaque étape
Workflow complet d'un SafeRider	Démarrer un timer, envoyer des positions, envoyer un audio, puis arrêter le timer
Gestion des tags	Créer un tag, l'associer à un SafeRider, puis le désassocier et le supprimer
Récupération des informations complètes d'un SafeRider	Tester get_saferider et vérifier que toutes les données (coordonnées, audios, tags) sont correctement retournées
Modification du profil utilisateur	Mettre à jour les informations personnelles et le message personnalisé, puis vérifier via get_account
Tentative de démarrage sans prérequis	Tester les 5 cas d'erreur possibles lors du start_timer (pas de contact, pas de message, etc.)
Suppression d'un SafeRider	Supprimer un SafeRider et vérifier que les fichiers (CSV, audios) et les entrées en base sont bien supprimés

Test sur le client

Pour valider ce projet, j’ai également effectué des tests du côté client. Pour mettre en place des test sur Kotlin, il suffit de les écrire dans le dossier : *app/src/test/java/com/okino813/sentinelle/ExampleUnitTest.kt.*

A l’intérieur de ce fichier, On définit une classe dans laquel je liste les fonctions de test. Pour les déclaré comme tel, j’utilise le décorateur « @Test » avant de déclarer la fonction. Ce fichier est généré lors de la création du projet. Il contient les import de Junit et de Mock.

Junit est un framework de tests unitaires pour Java. Mock quand à lui, sert à simuler le comportement d'objets pour pouvoir tester du code sans dépendre de son environnement réel.

Le fonctionnement des tests sur Kotlin est très similaire à Django. Voici pour exemple, un test unitaire qui vérifie la bon fonctionnement de la génération du fichier GPX :

```
class ExampleUnitTest {
    // Test unitaire
    @Test
    fun TU_008_vérification_generation_gpx() {
        // On instancie api_service avec un context mocké
        val mockContext = mockk<Context>(relaxed = true)
        val service = api_service(mockContext)

        val fakeSaferider = listOf(
            Saferider(
                id = 1,
                path = "",
                start_date = "2024-01-15T10:00:00",
                theorotical_end_date = "2024-01-15T11:00:00",
                real_end_date = "",
                locked = false,
                status = 1
            )
        )

        val fakeCoords = listOf(
            Triple(48.8566, 2.3522, 1700000000.0), // Paris
            Triple(43.2965, 5.3698, 1700000100.0)  // Marseille
        )

        val gpx = service.generateGpx(fakeSaferider, fakeCoords)

        assertTrue(gpx.contains("<trkpt lat=\"48.8566\" lon=\"2.3522\">"))
        assertTrue(gpx.contains("<trkpt lat=\"43.2965\" lon=\"5.3698\">"))
        assertTrue(gpx.contains("<trkseg>"))
        assertTrue(gpx.startsWith("<?xml"))
    }
...
```

Mise en production

Une fois le projet terminé, il ne me restait plus qu'à le déployer. Le projet, étant en deux parties, le déploiement ne sera pas de la même façon.

Déploiement de l'API

Comme pour tous mes projets que j'ai pu réaliser, je passe systématiquement par un étape lors de la réalisation du projet. Je crée un repository sur github et je l'upload dessus. Cela me permet d'envoyer mes modifications et de garder un système d'historique de version. Il est également possible, avec Github, de créer un workflow permettant le déploiement automatique sur un serveur. Pour cela, il est nécessaire de créer un dossier à la racine du projet. Ce dossier est .github/workflows/. Puis à l'intérieur, j'ai ajouté le fichier deploy.yml.

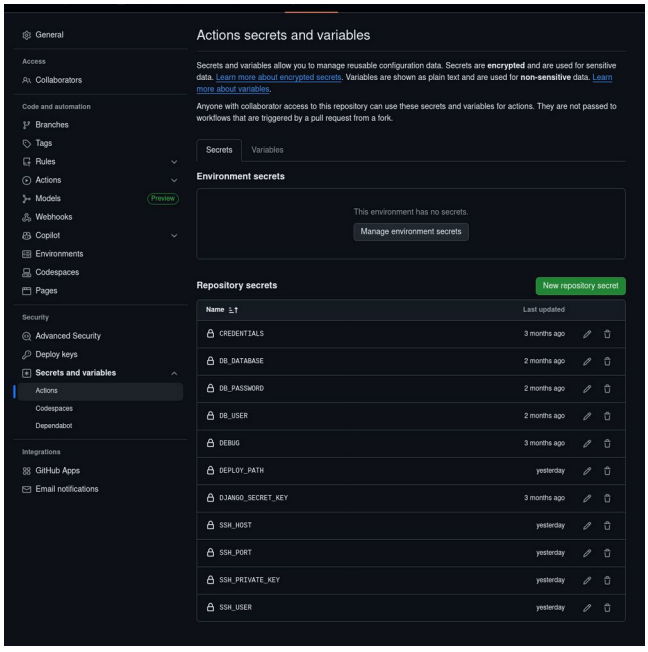
Pour déployer un repository, il existe deux façons. Soit on utilise « *Github-hoster runners* » ou un « *Self-hosted runners* ». La différence entre les deux, est la machine que l'on va utiliser pour lancer le déploiement. Le Github-hoster va utiliser une machine virtuelle fournie par Github. Le Self-hosted, va utiliser la machine qu'on va lui fournir. Pour le déploiement de mon projet, j'utilise un Github-hoster. Dans mon fichier deploy.yml, je définis mon choix avec les lignes suivantes :

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    needs: test
```

Ici, je spécifie que le runner doit être une machine virtuelle Ubuntu. Ensuite, je définis les variables d'environnement suivantes :

```yaml
env:
  SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
  SSH_HOST: ${{ secrets.SSH_HOST }}
  SSH_USER: ${{ secrets.SSH_USER }}
  SSH_PORT: ${{ secrets.SSH_PORT || 22 }}
  DEPLOY_PATH: ${{ secrets.DEPLOY_PATH || '/home/django/app' }}
```

Ces variables correspondent aux identifiants SSH de mon serveur. Pour des raisons de sécurité, les informations ne sont pas insérées en clair dans le fichier de déploiement. Elles sont enregistrées dans ce qu'on appelle des Secret Variables. Ce sont des variables contenues dans les paramètres du repository Github.



Une fois les variables configurées, il me faut définir les étapes pour déployer mon projet grâce à un simple push. Malheureusement pour moi, Github ne propose pas de deploy.yml clé en mains pour Django. J'ai donc été obligé de le créer. Pour ce faire, j'ai listé les grandes étapes nécessaires au déploiement.

Configuration SSH	Défini la clé de déploiement pour la machine virtuelle de Github. Vérifie si la structure des dossiers du serveur est en place pour accueillir le repo
Sauvegarde des fichiers de données	On déplace le dossier media, le .env et les fichiers log dans un dossier temporaire pour éviter de les remplacer
Mise à jour du code	Initialisation du repo sur le serveur et mise à jour du code
Restauration des fichiers de données	On restaure le dossier media, le .env et les fichiers log
Initialisation du VirtualEnv	Installation des modules via pip, inscrits dans le requirements.txt. Publication des assets et ressources statiques
Redémarrage des services	Redémarrage de Gunicorn et de Nginx

De cette façon, chaque changement qui a été fait sur la branche main, lancera de manière automatique le déploiement vers le serveur.

Conformément au standard de déploiement, tout nouvelle fonctionnalité ou correctif de bug, doit être fait sur une nouvelle branche portant le nom de la modification. Par exemple, pour l'ajout d'une fonctionnalité, je me rends sur la branche principale, la branche main :

```
git checkout main
```

Ensuite je crée une nouvelle branche avec ce nom :

```
git branch Feature/12FS-ajout-tag
```

Le nom de la branche doit suivre une convention. Elle doit commencer par le type de modification que l'on souhaite faire. Dans mon cas, c'est un ajout de fonctionnalité. Alors on commence le nom de la branche par « Feature/ ». A cela j'ajoute le nom du ticket que je suis en train de réaliser. Pour finir, séparé par des tirets, une courte description de la fonctionnalité.
Lorsque les modifications ajoutées fonctionnent, je fais un commit et un push sur cette branche.

```
git commit -m "Ajout de la requête d'ajout de contact"
```

```
git push origin Feature/12FS-ajout-tag
```

Puis je retourne sur la branche main, et je fusionne la nouvelle branche sur main.

```
git checkout main

git merge Feature/12FS-ajout-tag
```

De cette façon, le script de déploiement s'exécutera et s'enverra sur le serveur lorsque le merge aura été effectué.

Avant que les modification soit déployer, il est primordiale de tester l'API afin de s'assurer que nos modification n'ont pas eu d'incidence néfaste sur le bon fonctionnement de l'API. Pour cela, j'ai défini un environnement de test, s'exécutant avant le job deploy

```yaml
test:
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v4

    - name: Configurer Python
      uses: actions/setup-python@v5
      with:
        python-version: "3.11"

    - name: Créer l'environnement de test
      run: |
        python3.11 -m venv venv-test
        source venv-test/bin/activate
        pip install --upgrade pip --quiet
        pip install -r requirements.txt --quiet

    - name: Lancer les tests
      env:
        DJANGO_SETTINGS_MODULE: "sentinelle.settings"
        SECRET_KEY: ${{ secrets.SECRET_KEY }}
        DEBUG: "True"
        CREDENTIALS: "/tmp/firebase_credentials.json"
        DJANGO_MEDIA_ROOT: "/tmp/media_test"
      run: |
        echo '${{ secrets.FIREBASE_CREDENTIALS }}' > /tmp/firebase_credentials.json
        mkdir -p /tmp/media_test
        source venv-test/bin/activate
        python manage.py test --verbosity=2
```

Il faut que les tests soit tous valide pour pouvoir déployer. J'ajoute donc cette condition dans le job deploy de cette façon :

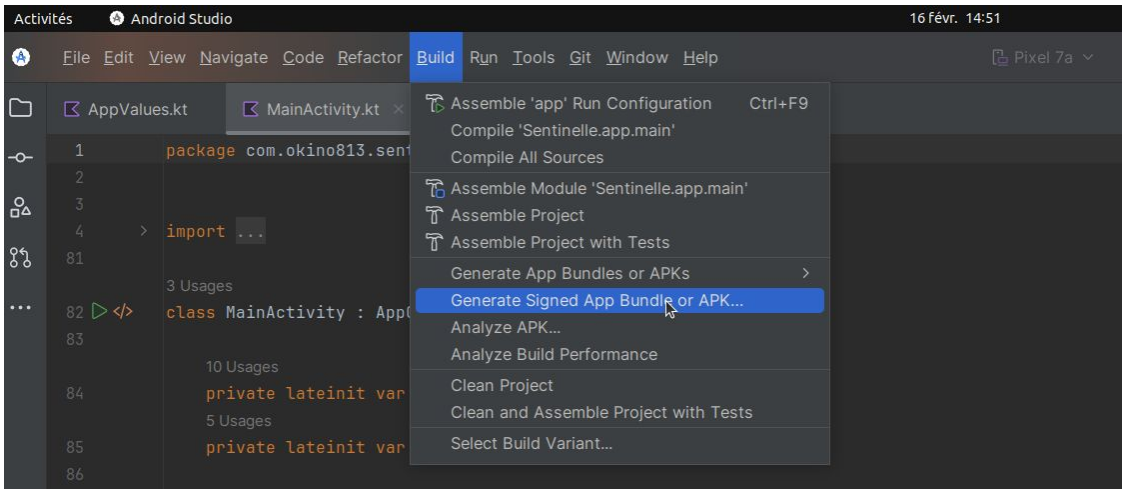
```yaml
deploy:
  runs-on: ubuntu-latest
  needs: test
```

Déploiement du Client

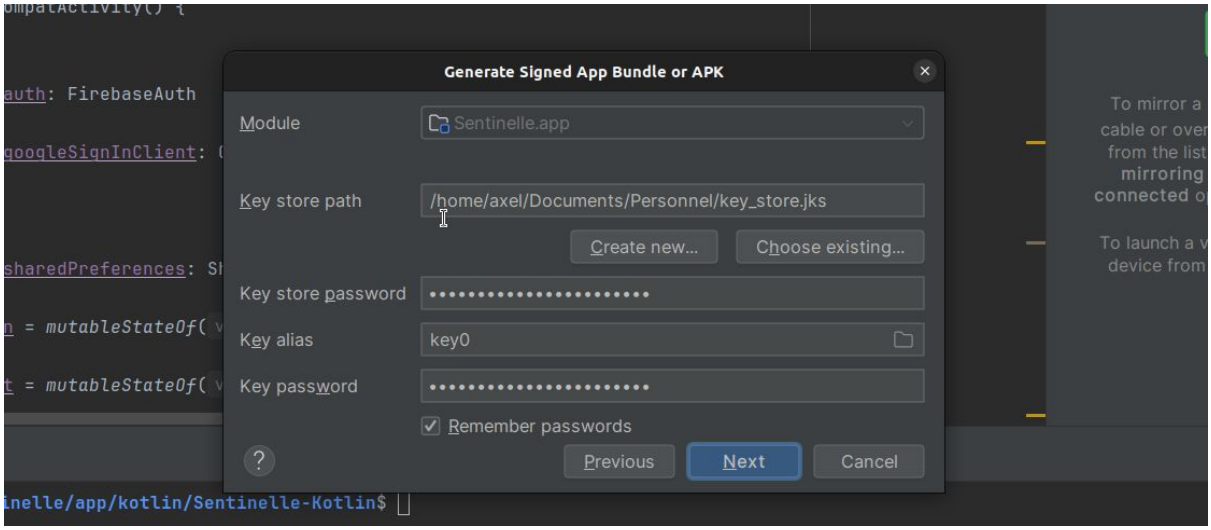
Le déploiement du client est complètement différent de l'API. En effet mon client est une application mobile qui doit être publiée sur Google Play Store. Pour se faire il est nécessaire de se créer un compte Google Developers. Cette inscription à un coût : 30€.

Une fois le compte créé, Je dois remplir plusieurs formulaires, intégré des liens de suppression de données, précisé ce que je prévois de faire pour chaque donnée récoltée et plein d'autres questions à répondre concernant l'application.

Pour publier une application, il faut fournir le fichier compilé qui sera téléchargé par les utilisateurs, à Google. Ce fichier doit être signé avec une clé pour s'assurer de l'intégrité du contenu de l'application. Sur Android Studio, cela se fait dans l'onglet « Build »



Un onglet nous propose d'exporter en « APK » ou en « App Bundle ». Dans mon cas, Google n'accepte que « App Bundle ». Puis un dernier onglet apparaît.



Je dois définir un « Key Store » et un mot de passe. Le fichier key_store.jks stocke les clée cryptographiques utilisées pour signer numériquement le fichier qui sera

compilé. En plus de ce fichier, un mot de passe doit être entré pour valider la signature. De cette manière, même en cas de fuite du Key Store, une personne malveillante ne pourra publier de version frauduleuse. En revanche, la perte du Key Store ou du mot de passe bloque définitivement les mise à jour. C'est pourquoi j'ai copié ce fichier à plusieurs endroits différents et que le mot de passe n'ai pas le même que mes autres mots de passe et est stocké dans un gestionnaire crypté.

Pendant longtemps, Google imposait aux développeur d'utiliser des pistes de test (test ouvert, fermé ou test interne) avec 100 utilisateurs. Cette obligation n'est plus d'aujourd'hui, mais reste tout de même fortement recommandée. L'inconvénient est qu'il faut que je trouve 100 utilisateurs pour tester l'application.


Nouveaux app bundles

Type de fichier	Version	Niveaux d'API	SDK cible	Formats d'écran	ABI	Fonctionnalités requises
App bundle	3 (1.0.2)	24 ou version ultérieure	35	4	4	4

Aujourd'hui je me retrouve bloqué car Google Play me demande des vidéos d'explications pour justifier la collecte de la localisation et du microphone. Par manque de temps je n'ai pas encore finalisé l'inscription. Les démarches à faire sont longues et très minutieuses.

Conclusion

Réaliser ce projet afin de pouvoir me présenter au passage du titre professionnel de concepteur développeur d'application a été pour moi l'occasion de découvrir de nouvelles technologies et de nouvelles méthodes de travail.

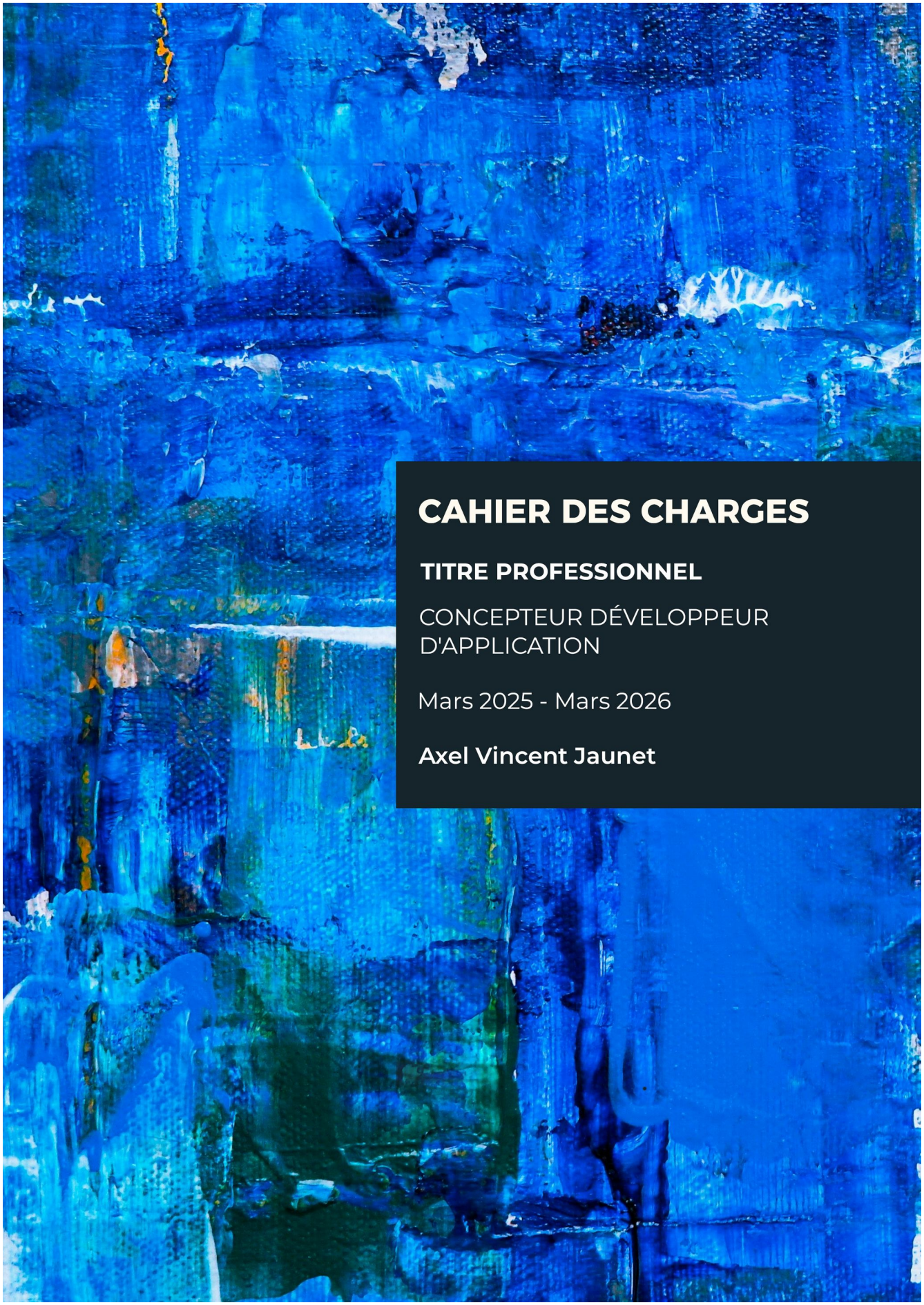
En effet, Kotlin et le développement mobile en général, ont été pour moi des découvertes et même si les débuts ont été très compliqués (surtout pour Jetpack Compose), j'ai fini par prendre beaucoup de plaisir à utiliser ces deux technologies. Ce projet m'a surtout permis de sortir de ma zone de confort de développeur. Par la suite je souhaiterais me spécialiser dans le développement Android.

Ensuite ce projet m'a rappelé qu'une bonne réflexion en amont permet toujours de gagner du temps par la suite. La réalisation d'un bon cahier des charges ainsi que les dossiers de conception fonctionnelle et technique est primordiale pour pouvoir commencer à coder son application.

J'ai voulu tester quelque chose de nouveau et j'ai pris plaisir à explorer ce qu'est le développement Android. Je suis globalement satisfait du résultat de ce projet et cela

m’a apporté plusieurs idées de projet que je ferais dès la fin de mon titre professionnel.

Annexe



CAHIER DES CHARGES

TITRE PROFESSIONNEL

CONCEPTEUR DÉVELOPPEUR D'APPLICATION

Mars 2025 - Mars 2026

Axel Vincent Jaunet

Sommaire

Présentation du projet

Contexte du projet

Sentinelle est née d'un constat personnel. À chaque fois que je sortais de chez moi, je me sentais obligé de prévenir un proche de mon trajet ou de mon arrivée, une habitude rassurante mais rapidement contraignante. Un jour, en me rendant à un rendez-vous incertain, à une remise en main propre avec un inconnu via Leboncoin, j'ai réalisé à quel point cette précaution, bien que nécessaire, était fastidieuse à répéter. De plus, elle ne garantissait pas une réaction rapide en cas de problème réel.

C'est de cette frustration et de ce besoin de sécurité simplifiée que m'est venue l'idée : et si une application pouvait automatiser cette vigilance ? Plutôt que d'envoyer manuellement un message à chaque déplacement, Sentinelle permet de définir un minuteur. Si je n'annule pas ce compte à rebours à mon arrivée (ou si un incident survient), mes proches reçoivent automatiquement ma localisation et des enregistrements audio, sans que j'aie à faire quoi que ce soit. Ainsi, je peux me déplacer sereinement, même dans des situations potentiellement risquées, en sachant que mes proches seront alertés seulement si nécessaire.

Au-delà de mon usage personnel, ce système répond à un besoin universel : allier sécurité et simplicité, pour que chacun puisse vaquer à ses occupations sans la charge mentale de devoir constamment rassurer son entourage. Sentinelle n'est pas qu'une application, c'est la concrétisation d'une solution pensée pour ceux qui, comme moi, refusent de choisir entre liberté et sécurité.

Objectifs du projet

L'objectif de ce projet est de mettre en place une application mobile sur Android pour mon usage personnel pour tester, et ensuite de le proposer sur Google Play Store. Le projet devra avoir un site web pour présenter l'application et pouvoir afficher le trajet d'un utilisateur grâce à un lien spécifique.

Aucun volume de téléchargement n'est visé. Toutefois, l'application devra contenir n pages dans la partie application mobile :

- Une page d'accueil où se situera le minuteur et un bouton d'alerte manuel.
- Une page qui contient la liste des trajets effectués
- Une page qui détail un trajet
- Une page pour définir un message personnalisé
- Une page pour ajouter des contacts, les sélectionner et supprimer
- Une page de création et de suppression de Tag personnalisé

- Une page paramètre du compte utilisateur ainsi qu'un bouton mode contraster, suppression des trajets, suppression du compte, déconnexion.
- 4 pages de didacticiel

Au niveau du front-office du site internet :

- Une page de politique de confidentialité
- Une page de mention légales
- Une page d'accueil de présentation du projet et de formulaire de contact

Le site comportera également une partie « back office » ou seule les personnes autorisées pourront se rendre via une connexion sécurisée. Il y sera possible de gérer (création, modification suppression) les utilisateurs, les tag, les trajets, contacts et messages personnalisés.

Les cibles

La partie « front-office » a pour cible toutes les personnes souhaite se rassurer eu ou leurs proches. Cela comprends :

- Travailleur isolé
- Écolier
- Piétons
- Conducteur
- Touriste

Le back-office, seul l'administrateur (moi) pourra y accéder.

Périmètre du Projet

Le projet sera fait pour un public français. En effet ce projet n'a pas la vocation d'être utilisé à grande échelle. En effet l'envoi de numéro de téléphone ne concerne que la France. Pour une utilisation dans plusieurs pays, cela engendre plus de coûts.

Fonctionnalités à développer

Les fonctionnalités principales du projet seront les suivantes :

- Inscription utilisateur
- Connexion utilisateur

- Lancer un minuteur

    Enregistrement de la localisation et de l'environnement sonore durant la durée du minuteur, s'il est écoulé, les enregistrements sont envoyé un proche

- Alerte manuelle

    En plus du minuteur, l'utilisateur pourra également déclencher une alerte manuelle

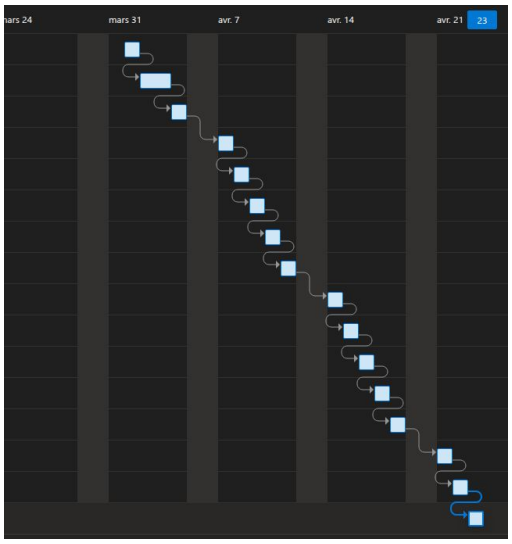
- Ajouter des contacts
- Définir un message personnalisé
- Lister les trajets

Planification

Délais / planning prévisionnel

Pour réaliser ce projet, j'ai commencé par lister les fonctionnalité à développer et je leurs ai attribué un temps pour chaque fonctionnalités

Nom de la tâche	Début	Dép...	Attribu...	D...
Création du projet	1/4/2025		Axel	1 jour
Authentification Firebase	2/4/2025	1FS	Axel	2 jours
Création des models	4/4/2025	2FS	Axel	1 jour
Register nouveau utilisateur	7/4/2025	3FS	Axel	1 jour
Crud contact	8/4/2025	4FS	Axel	1 jour
Crud Saferider	9/4/2025	5FS	Axel	1 jour
Envoi des coordonées GPS	10/4/2025	6FS	Axel	1 jour
Envoie des fichiers audio	11/4/2025	7FS	Axel	1 jour
Supression du compte et des données	14/4/2025	8FS	Axel	1 jour
Supression de tout les trajets	15/4/2025	9FS	Axel	1 jour
Cron fin minuteur	16/4/2025	10FS	Axel	1 jour
Envoi du SMS	17/4/2025	11FS	Axel	1 jour
Ajout du Tag	18/4/2025	12FS	Axel	1 jour
Assignation du tag	21/4/2025	13FS	Axel	1 jour
Afficher trajet sur Navigateur	22/4/2025	14FS	Axel	1 jour
Mode contraster	23/4/2025	15FS	Axel	1 jour



De cette façon, j'ai pu faire un plan grâce à l'outil « Planificateurs » de Microsoft Teams. Le calendrier étant configuré pour un travail à temps pleins, je ne pouvais pas m'en servir pour connaître une date précise de fin de développement. Cependant j'ai pu déterminer le développement allait me prendre deux semaines. En le croisant avec mon planning de formation, j'ai pu déterminer que j'allais finir le projet en décembre 2025.

Échéances importantes

La date butoir de mon projet est le jours du passage de l'examen. Cependant je souhaiterais terminer ce projet avant la fin de l'année 2025.

Contraintes

Contraintes techniques

Pour la réalisation de ce projet, je dois obligatoirement utiliser une architecture « n-tiers ». Cela signifie qu'une API devra être mise en place, ainsi qu'une interface utilisateur.

Cette interface utilisateur devra être utilisée sur un téléphone Android. Pour maximiser les performances, le langage devra être le langage natif Kotlin.

Au niveau de l'API, encore une fois pour maximiser les performances, le framework Django a été choisi pour sa légèreté.

Contraintes réglementaires

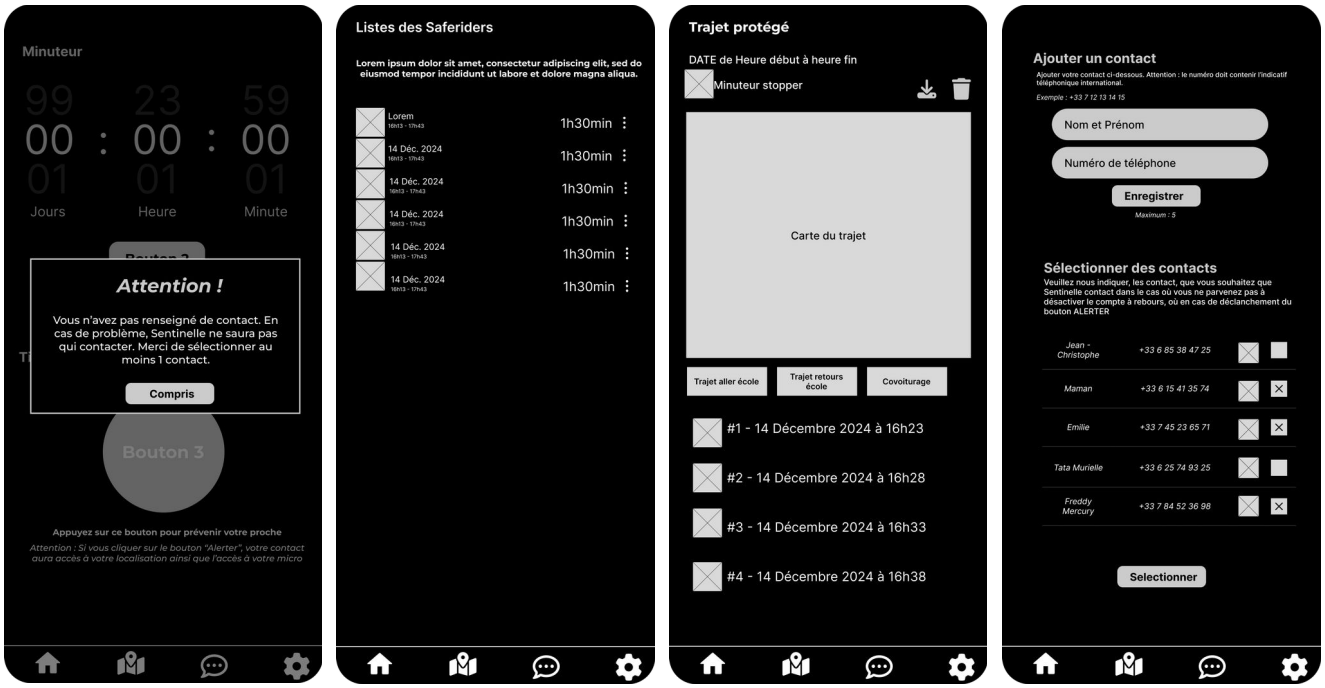
Pour respecter le RGPD, une politique de confidentialité des données sera mise à disposition sur le site web et devra être acceptée pour créer un compte utilisateur. Les données récolté ne doivent être utilisé que pour le fonctionnement de l'application et ne pas servir à démarcher ou à être vendu à un tiers.

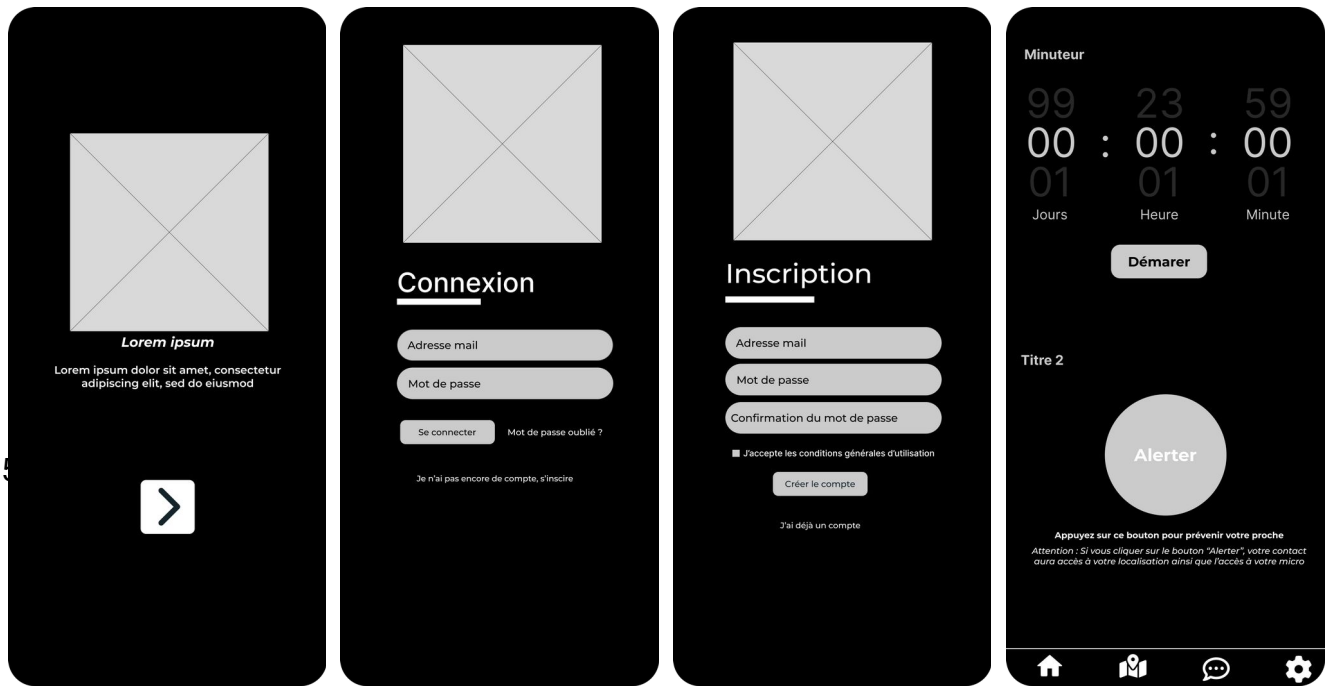
Pour l'accessibilité, les couleurs devront avoir un score de contraste suffisant. De plus, un mode contraste devra être mis en place, activable et désactivable facilement par l'utilisateur.

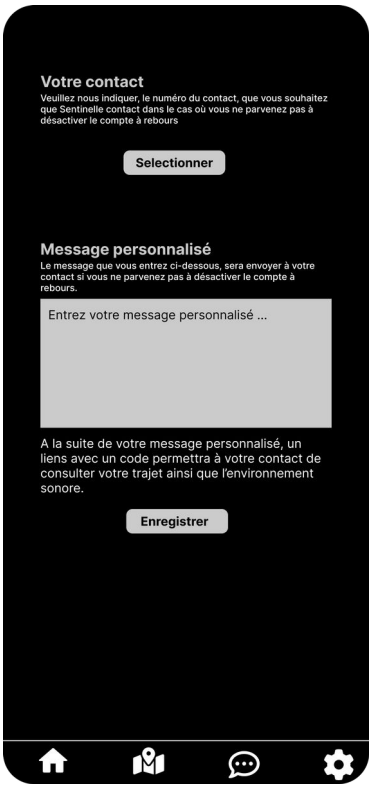
Design & UX/UI

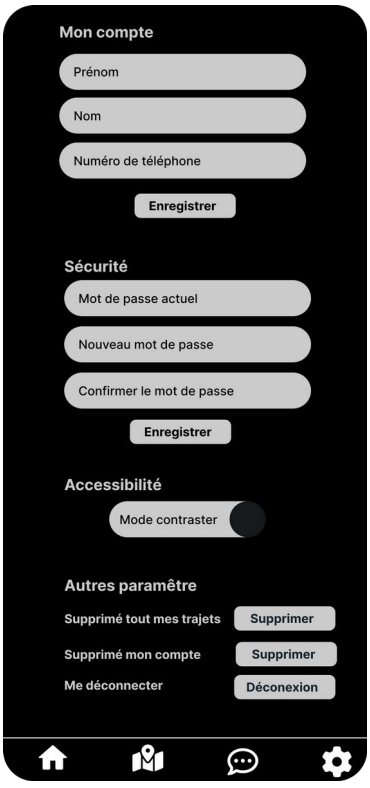
Maquettes fonctionnelles (wireframes)

Les wireframes des templates de l’application :

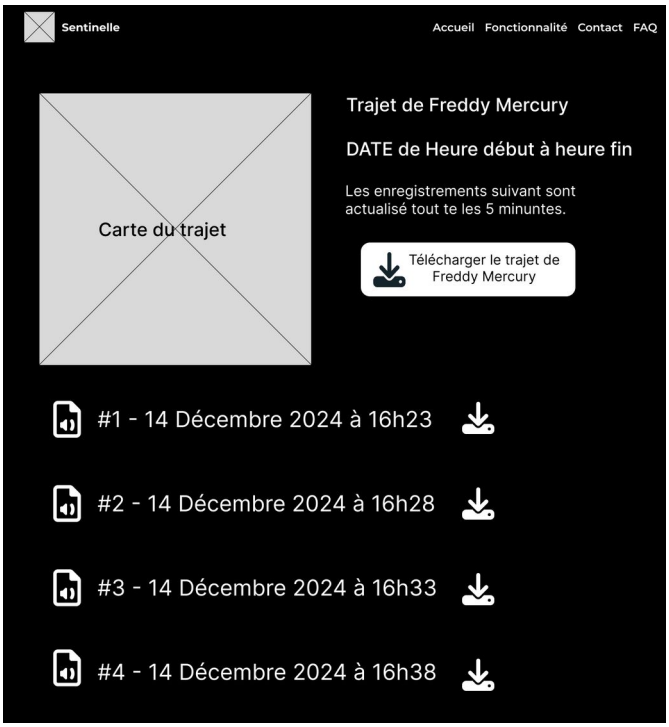






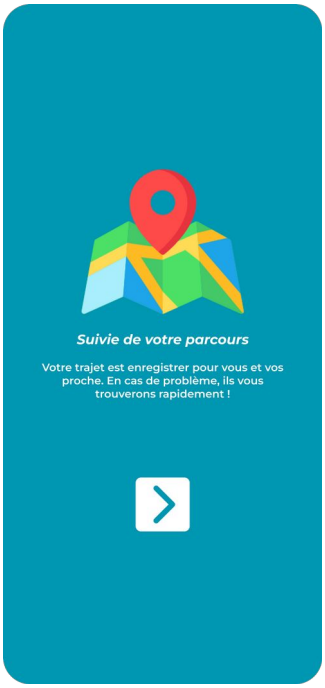


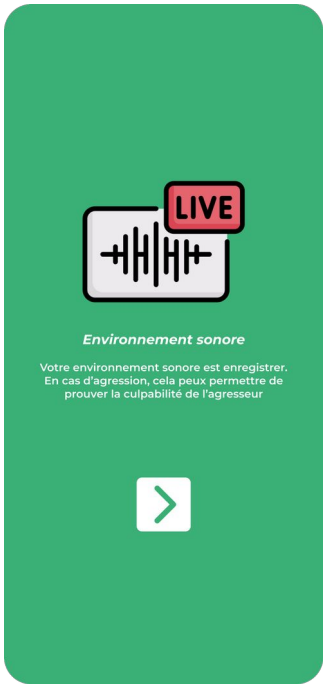
Et pour l'affichage d'un trajet sur le site web :

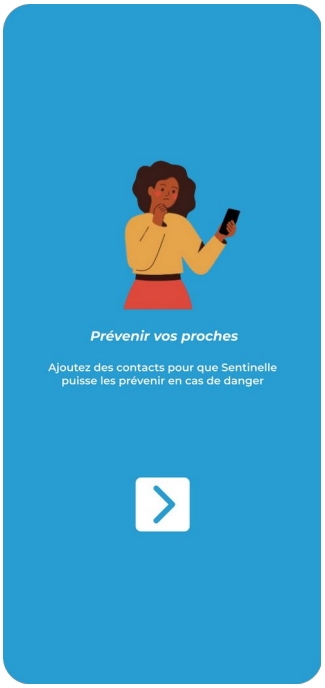


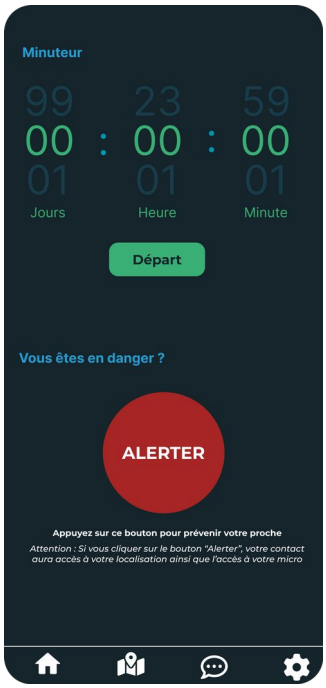
Maquettes graphiques

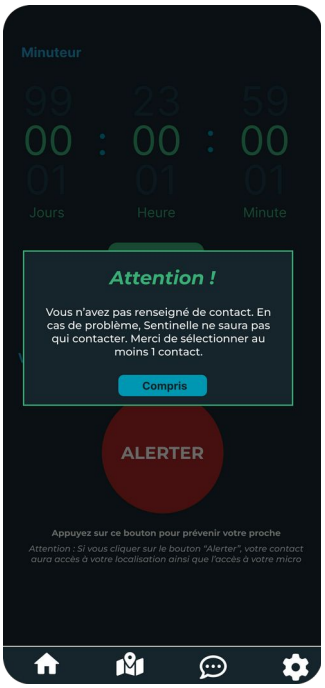
Les maquettes des wireframes avec les images et la charte graphique ajouter :

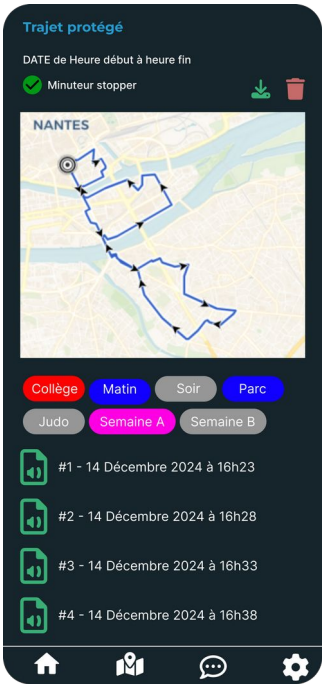


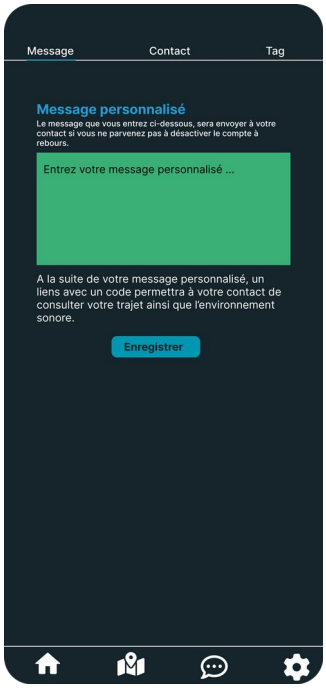


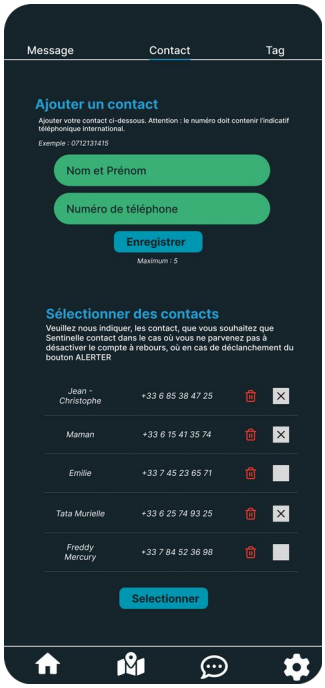


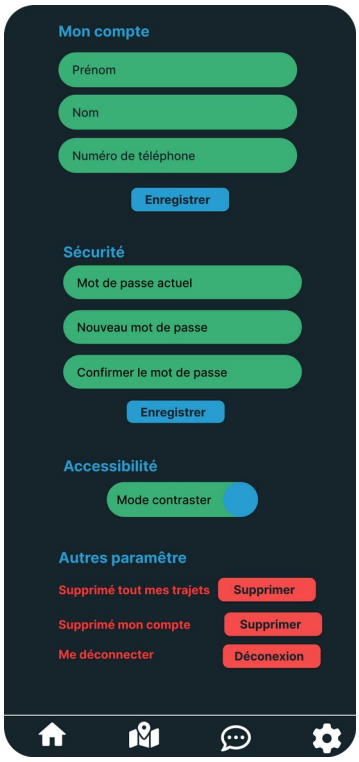








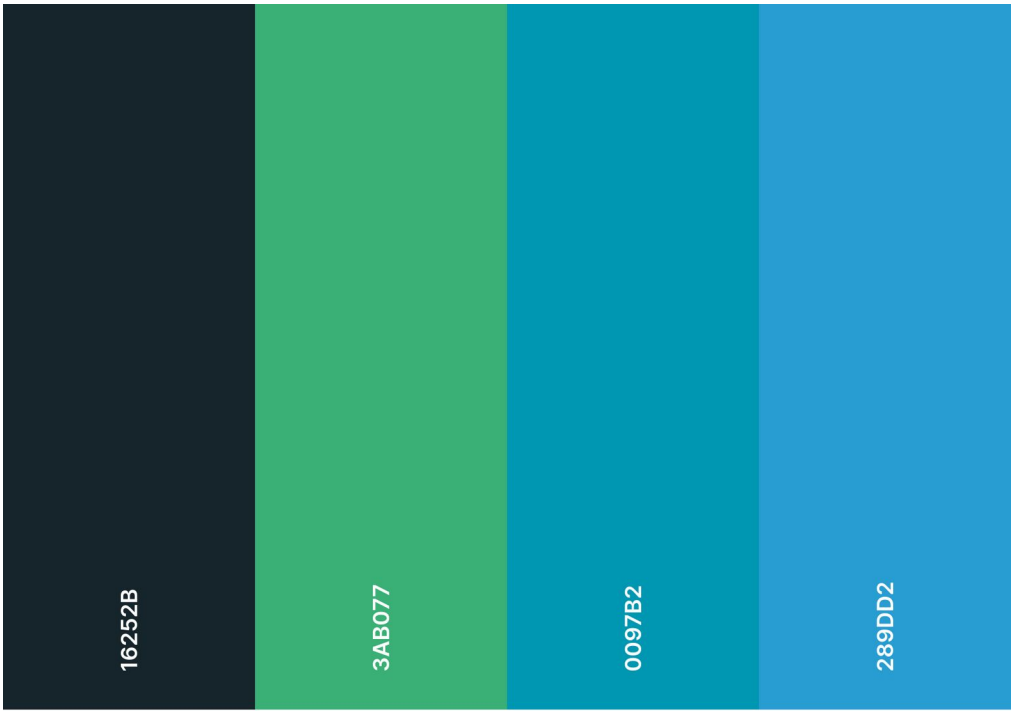




Charte graphique

Les qualités qu'inspire ce projet sont : La bienveillance et la sécurité. Les couleurs principales sont le bleu et le vert.

Palette de couleurs



Police d'écriture

Montserrat

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Normal**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Bold**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Italic**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Bold**
Style : **Italic**

Roboto

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Normal**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Bold**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Regular**
Style : **Italic**

Aa
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
1234567890()#@
Graisse : **Bold**
Style : **Italic**

Logo et déclinaisons

Le développement

La mise en place de ce projet se basera sur l'architecture à couches « n-tiers », ce qui permettra d'utiliser l'application mobile ainsi que le site web.

L'application mobile sera réalisée avec Kotlin et Jetpack Compose, avec une version du JDK assurant la plus grande compatibilité des téléphones Android : le JDK 24 pour Android 7 (Nougat).

Le site web ainsi que l'API REST seront élaboré avec le framework Django. Sous la version 3 de Python. La gestion des données se fera avec le SGBD MySQL

Dépôts du nom de domaine

Le dépôt du nom de domaine du site web sera « sentinelle-app.net » et devra être renouvelé par le concepteur du site.

L’hébergement

L’API sera mis en ligne sur un serveur OVH. Le client sera disponible sur le catalogue d’application Google Play Store.

SPÉCIFICATIONS FONCTIONNELLES

TITRE PROFESSIONNEL

CONCEPTEUR DÉVELOPPEUR D'APPLICATION

Mars 2025 - Mars 2026

Axel Vincent Jaunet

Sommaire

Introduction

Objectif du document

L'objectif de ce document est de présenter les fonctionnalités du système qui seront développées pour l'application Sentinelle. Les éléments composant ce dossier découlent du cahier des charges.

Références

Pour obtenir plus d'informations concernant les aspects techniques de ce projet, référez-vous au dossier de conception technique de l'application. Pour obtenir des détails sur les besoins des futurs utilisateurs, référez-vous au cahier des charges.

Contexte

Lorsque l'on sort de chez soi, il est nécessaire de prévenir ses proches du lieu où l'on se rend et de l'heure à laquelle on sera de retour. Ces préventions ne sont pas efficaces en cas de réel danger.

Enjeux et objectifs

L'objectif est de pouvoir se protéger face au danger en alliant la simplicité d'utilisation et la protection des données. En effet, l'application devra permettre de :

- Enregistrer les coordonnées et l'environnement sonore
- Prévenir les contacts de l'utilisateur en cas de danger grâce à un bouton manuel
- Prévenir les contacts de l'utilisateur en cas d'un minuteur écoulé

Description général du projet

Pour répondre à ces besoins, une application mobile sera développée (utilisation de Kotlin 1.9.23 avec Jetpack Compose) et d'une API REST + site web (utilisation du framework Django 5.2.9). L'application mobile ainsi que le site web devront être développés de façon à s'adapter à tout type d'écran et penser en termes d'UI Design afin que l'utilisateur bénéficie d'une expérience ergonomique du système. Ces critères-là seront menés à bien grâce à l'utilisation du framework Jetpack Compose

et de Material Design 3.

Les principes de fonctionnement

Le fonctionnement de ce système devra être adapté en fonction du type d'utilisateur, ce qui signifie qu'il y aura une interface dédiée aux administrateurs, une autre dédiée aux utilisateurs, une pour les proches de l'utilisateur et enfin une autre réservée aux simples internautes. Voici donc les 4 acteurs : Visiteurs, Proches, Utilisateurs et Administrateurs.

L'administrateur pourra se connecter à l'application grâce à son identifiant, qui sera son adresse e-mail et son mot de passe.

De manière générale, tout les internautes pourront consulter le site internet sans se connecter. Ce n'est que sur l'application mobile qu'une connexion sera demandée pour les utilisateurs.

Interface visiteur (Front-office site web)

Les visiteurs n'auront accès qu'à la page vitrine de l'application, aux mentions légales, à la politique de confidentialité et à la page de demande de suppression de compte.

Interface d'administration (Back-office site web)

L'accès à cette interface se fera uniquement après une authentification et une vérification de rôle. Les administrateurs seront automatiquement redirigés vers l'interface admin afin de pouvoir gérer les comptes, les trajets, les tags et les contacts des utilisateurs. Les administrateurs sont les seuls à pouvoir modifier le statut des utilisateurs.

Interface utilisateur (Application Mobile)

L'accès à cette interface se fera via le Google Play Store. L'utilisateur peut se créer un compte et s'y connecter. Il peut définir un minuteur, lancer une alerte manuelle, définir un message personnalisé, gérer son compte, ses contacts, ses tags et ses trajets.

Les fonctionnalités par acteurs

Le visiteur : Il ne peut que consulter les pages du site, contacter l'administrateur

grâce au formulaire de contact.

Le proche : Il peut consulter le trajet d'un proche

L'utilisateur : Il peut s'identifier et utiliser l'application mobile.

L'administrateur : Dispose de tout les droits. Il peut gérer tous les utilisateurs et leurs données.

Fonctionnalité : Gestion des utilisateurs

Fonctionnalité	Administrateur	Utilisateur	Visiteur
Ajouter un utilisateur	√	√	√
Se connecter	√	√	
Afficher son profil	√	√	
Modifier son profil	√	√	
Supprimer un utilisateur	√		
Afficher la liste de tous les utilisateurs	√		
Modifier son mot de passe	√	√	
Modifier le profil d'un autre utilisateur	√		
Définir un message personnalisé	√	√	

Fonctionnalité : Gestion des trajets

Fonctionnalité	Administrateur	Utilisateur	Visiteur
Lancer un minuteur		√	
Stopper un minuteur		√	
Modifier un trajet	√	√	
Supprimer un trajet	√	√	

Assigner un tag	√	√	
Voir un trajet	√	√	√
Afficher la liste des trajets	√	√	
Télécharger le trajets		√	√

Fonctionnalité : Gestion des Tags

Fonctionnalité	Administrateur	Utilisateur	Visiteur
Crée un tag	√	√	
Supprimer un tag	√	√	
Afficher la liste des tags	√	√	

Fonctionnalité : Gestion des contacts

Fonctionnalité	Administrateur	Utilisateur	Visiteur
Crée un contact	√	√	
Supprimer un contact	√	√	
Afficher la liste des contacts	√	√	

Fonctionnalité : Formulaire de contact

Fonctionnalité	Administrateur	Utilisateur	Visiteur
Afficher le formulaire			√
Envoyer un message			√

Diagrammes de cas d’utilisation

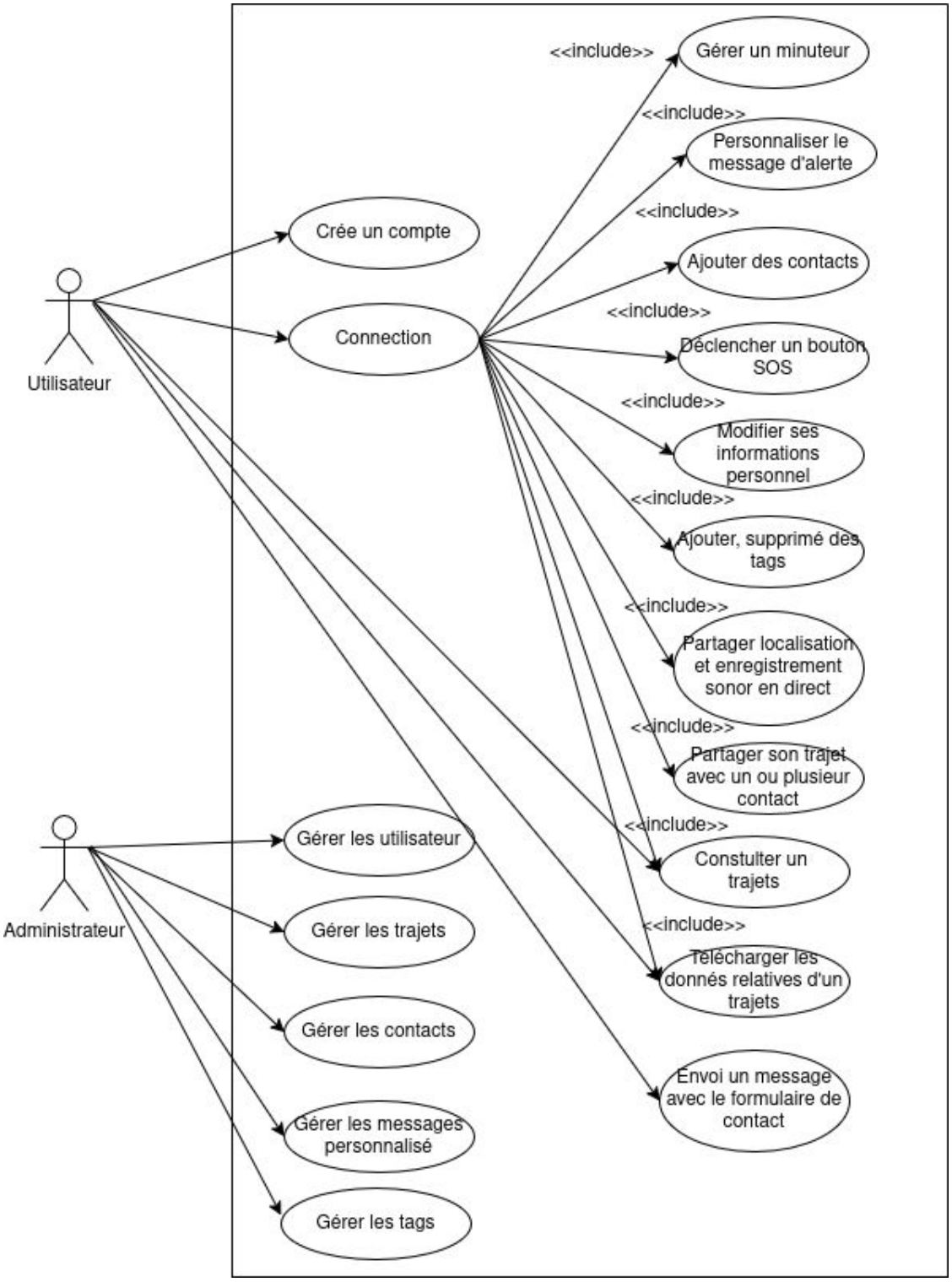


Diagramme d’activité

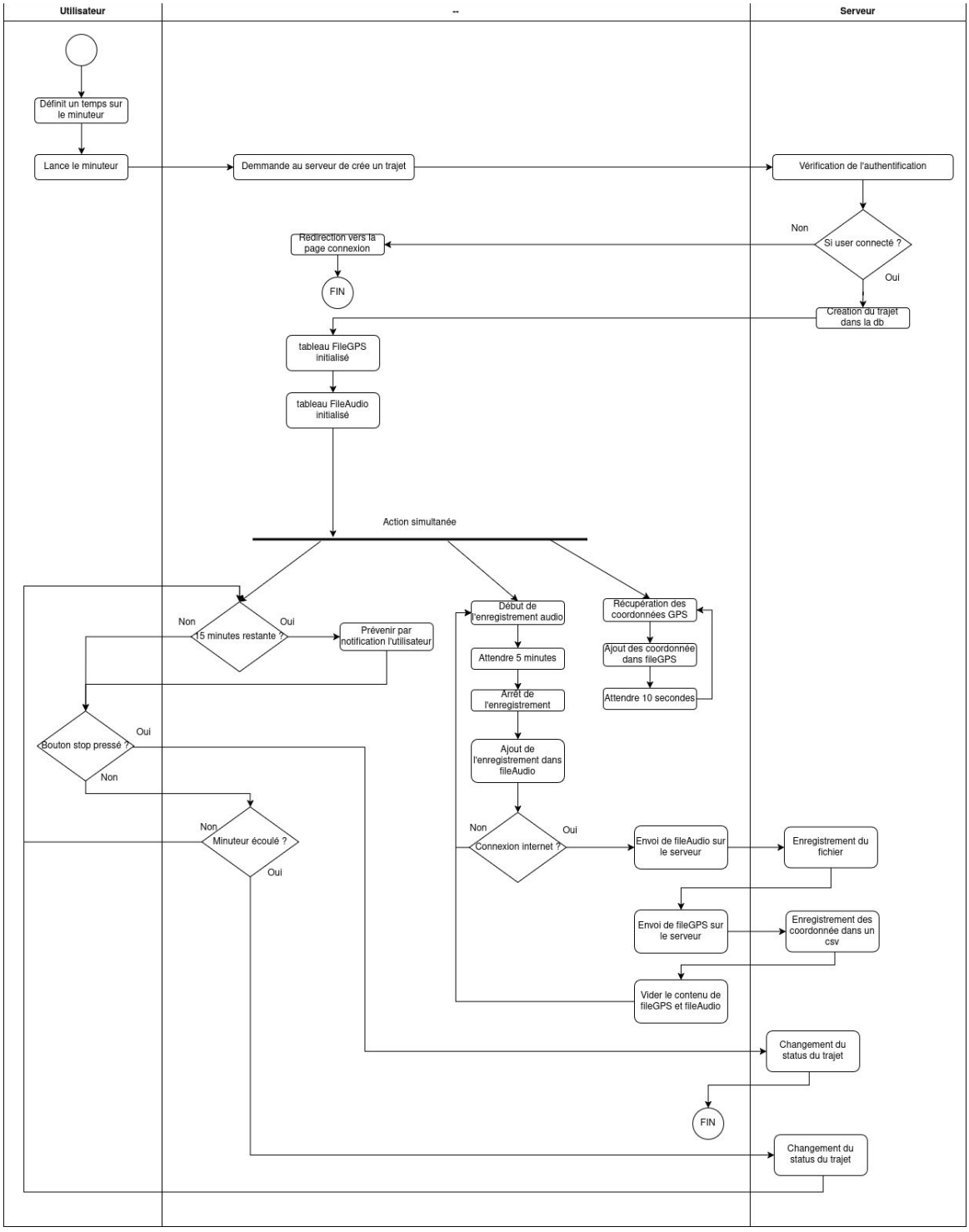


Diagramme d'activité du minuteur et de la récolte d'enregistrement

SPÉCIFICATIONS TECHNIQUES

TITRE PROFESSIONNEL

CONCEPTEUR DÉVELOPPEUR D'APPLICATION

Mars 2025 - Mars 2026

Axel Vincent Jaunet

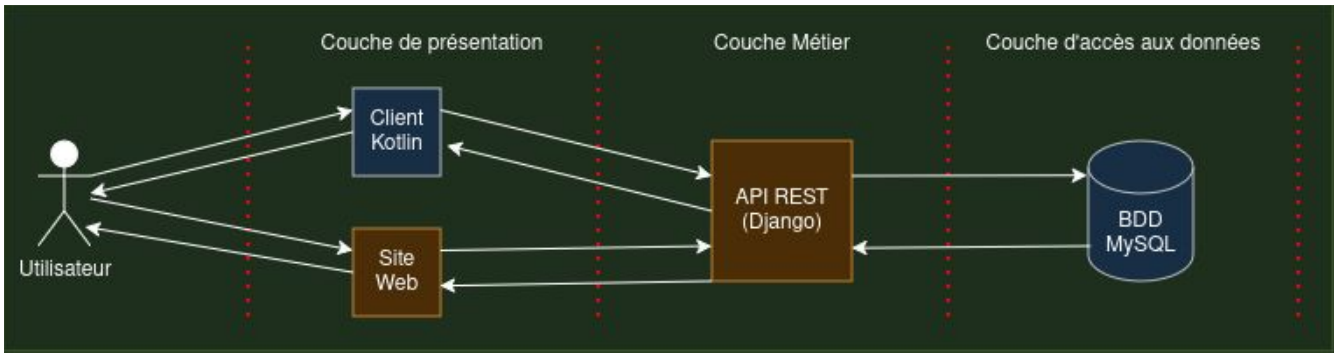
Sommaire

Objectif du document

Ce document constitue le dossier de conception technique de l'application Sentinelle. Les objectifs du document sont d'apporter plus de détails techniques. Ce document fait suite au cahier des charges et aux spécifications fonctionnelles.

Architecture technique

Pour pouvoir mettre en place une application mobile et une application web, il faudra utiliser l'architecture n-tiers. Sentinelle est une petite application, donc il n'y a qu'une seule base de données relationnelle avec MySQL.



Application Web et Mobile

Back-End

Le Back-end est la partie invisible du site en charge du coté logique de l'application mobile. Celui ci sera codé en Python avec le framework Django. Il sera en lien direct avec la base de données relationnelle.

Front-End

Le Front-End est la partie visible du site web. Elle comprends le Front-office et le Back-office. HTML, CSS et Javascript sera utilisé.

Application Mobile

L'application mobile sera développée pour les téléphones Android, au moyens du Framework Kotlin et Jetpack Compose, tout les deux conçu par Google, assurant de meilleures performance dans l'environnent Android.

Base de données

Le système de gestion de base de données choisi est MySQL. C'est un outil de gestion de base de données externes à l'API. Elle va contenir les informations nécessaires à toute les fonctionnalité de l'application, comme les trajets, utilisateurs, contacts, tags, enregistrement ….

Dictionnaire des données

Table User		
Nom	Type	Taille
id_user	AUTO_INCREMENT	
firstname	VARCHAR	50
lastname	VARCHAR	50
email	VARCHAR	50
firebase_uid	VARCHAR	128
custom_message	VARCHAR	150
phone	VARCHAR	10

Table Tag		
Nom	Type	Taille
id_tag	AUTO_INCREMENT	
name	VARCHAR	50

Table Saferider		
Nom	Type	Taille
id_saferider	AUTO_INCREMENT	
start_date	DATETIME	
theoretical_end_date	DATETIME	
real_end_date	DATETIME	
path	VARCHAR	50
lock	BOOLEAN	

Table Record		
Nom	Type	Taille
id_record	AUTO_INCREMENT	
date	DATETIME	
path	VARCHAR	50

Table Type		
Nom	Type	Taille
id_type	AUTO_INCREMENT	
name	VARCHAR	50

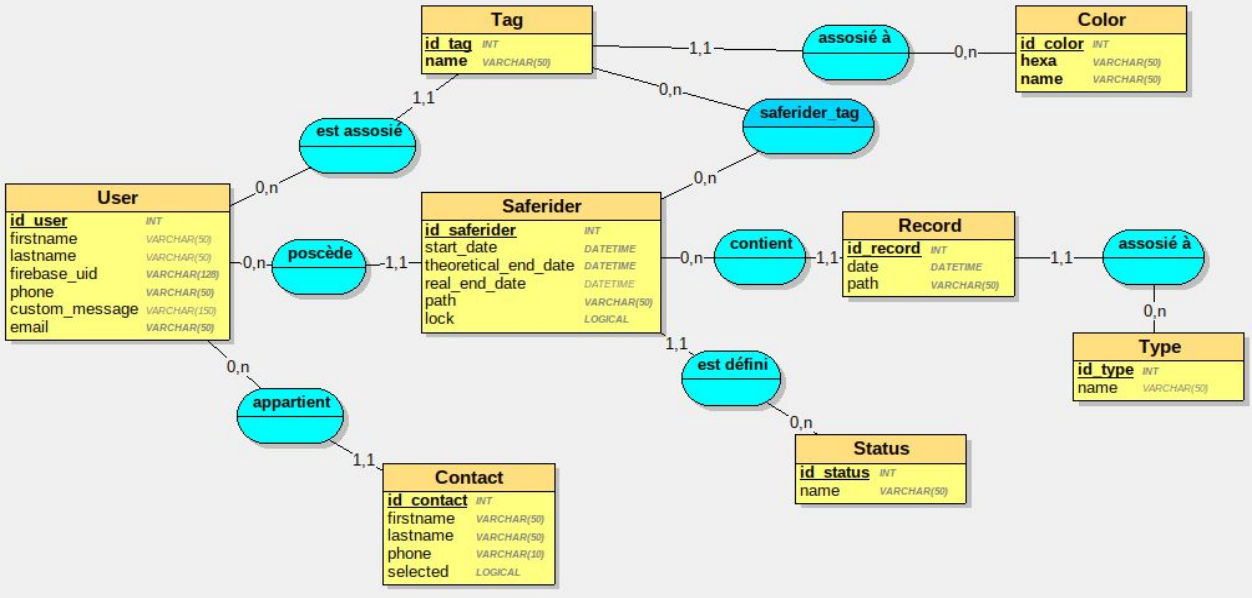
Table Color		
Nom	Type	Taille
id_color	AUTO_INCREMENT	
hexa	VARCHAR	50
name	VARCHAR	50

Table Status		
Nom	Type	Taille
id_status	AUTO_INCREMENT	
name	VARCHAR	50

Modélisation de la base de données

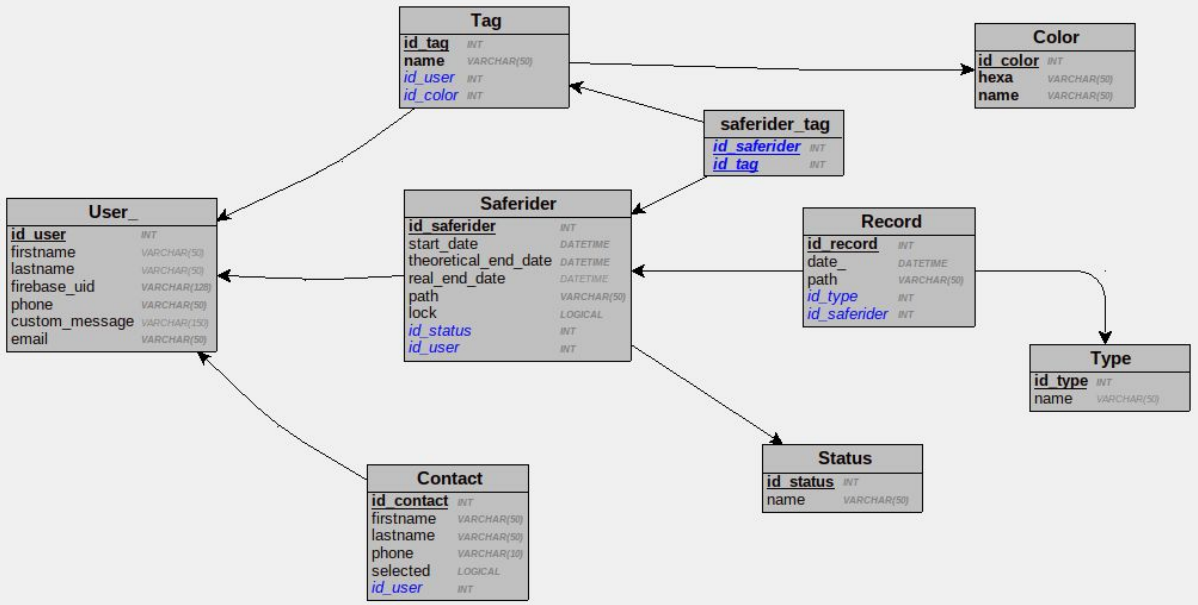
Le MCD

Le modèle conceptuel de base de données représente les donnée issues du dictionnaire, sous forme d'entité en ajoutant des détails sur les relations qui existe entre elles.



Le MLD

Le modèle logique des données est issu du MCD. Cependant ce ne sont plus des entité, mais des tables. Dedans sont représentées les clés primaires et étrangères



Détails des tables

Les caractéristiques des différentes tables sont rappelées ci-dessous ; on retrouve la table cible avec la précision des attributs. Chaque tableau comporte une colonne décrivant le type de l'attribut, son nom, sa définition, précise si l'attribut peut être nul ou non et l'illustre d'un exemple concret.

Saferider :

Nom

Saferider

Nom logique : Saferider

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_saferider	id_saferider	INT	✓	✓
	start_date	start_date	DATETIME	✓	
	theoretical_end_date	theoretical_end_date	DATETIME	✓	
	real_end_date	real_end_date	DATETIME		
	path	path	VARCHAR(50)	✓	
	lock	lock	LOGICAL	✓	

User :

Nom

User

Nom logique : User_

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_user	id_user	INT	✓	✓
	firstname	firstname	VARCHAR(50)		
	lastname	lastname	VARCHAR(50)		
	firebase_uid	firebase_uid	VARCHAR(128)	✓	
	phone	phone	VARCHAR(50)	✓	
	custom_message	custom_message	VARCHAR(150)		
	email	email	VARCHAR(50)	✓	

Tag :

Nom

Tag

Nom logique : Tag

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_tag	id_tag	INT	✓	✓
	name	name	VARCHAR(50)	✓	✓

Color :

Nom

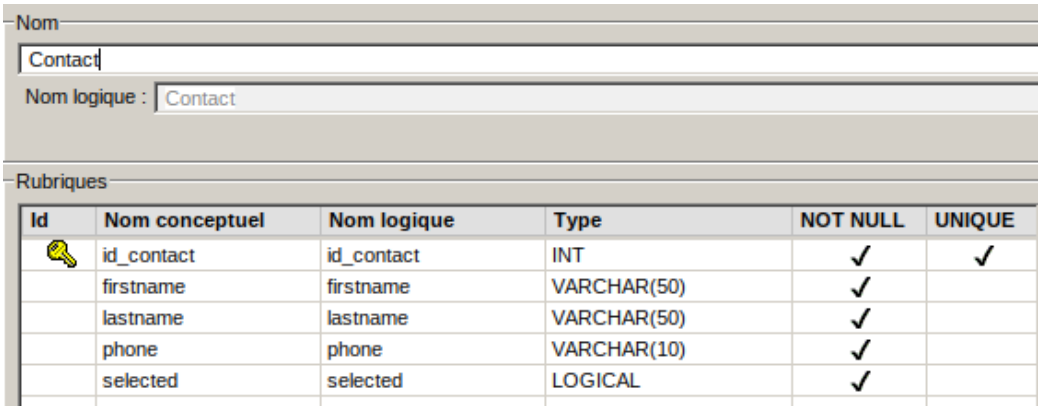
Color

Nom logique : Color

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_color	id_color	INT	✓	✓
	hexa	hexa	VARCHAR(50)	✓	✓
	name	name	VARCHAR(50)	✓	✓

Contact :



Nom

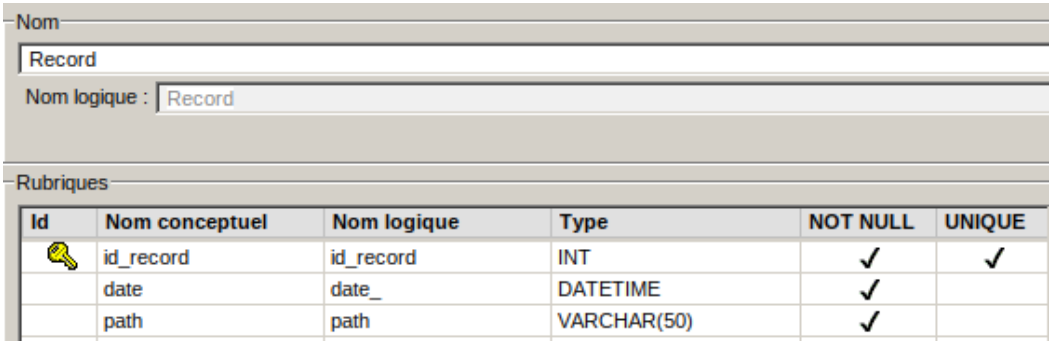
Contact

Nom logique : Contact

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_contact	id_contact	INT	✓	✓
	firstname	firstname	VARCHAR(50)	✓	
	lastname	lastname	VARCHAR(50)	✓	
	phone	phone	VARCHAR(10)	✓	
	selected	selected	LOGICAL	✓	

Record :



Nom

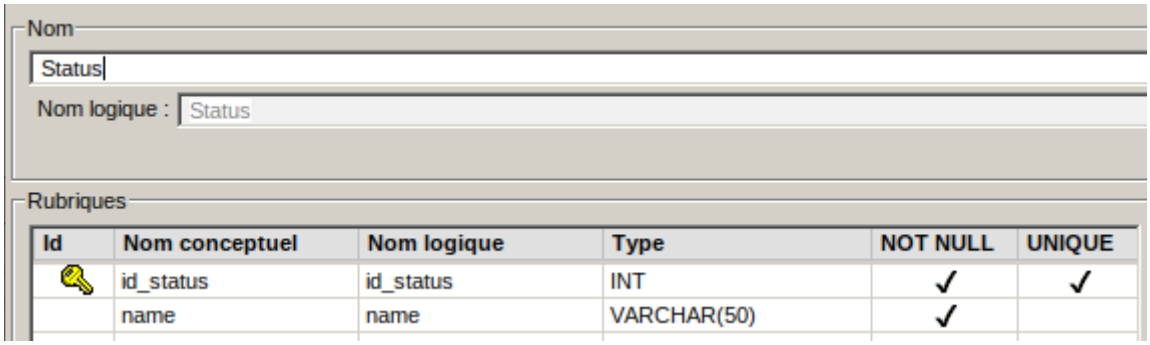
Record

Nom logique : Record

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_record	id_record	INT	✓	✓
	date	date_	DATETIME	✓	
	path	path	VARCHAR(50)	✓	

Status :



Nom

Status

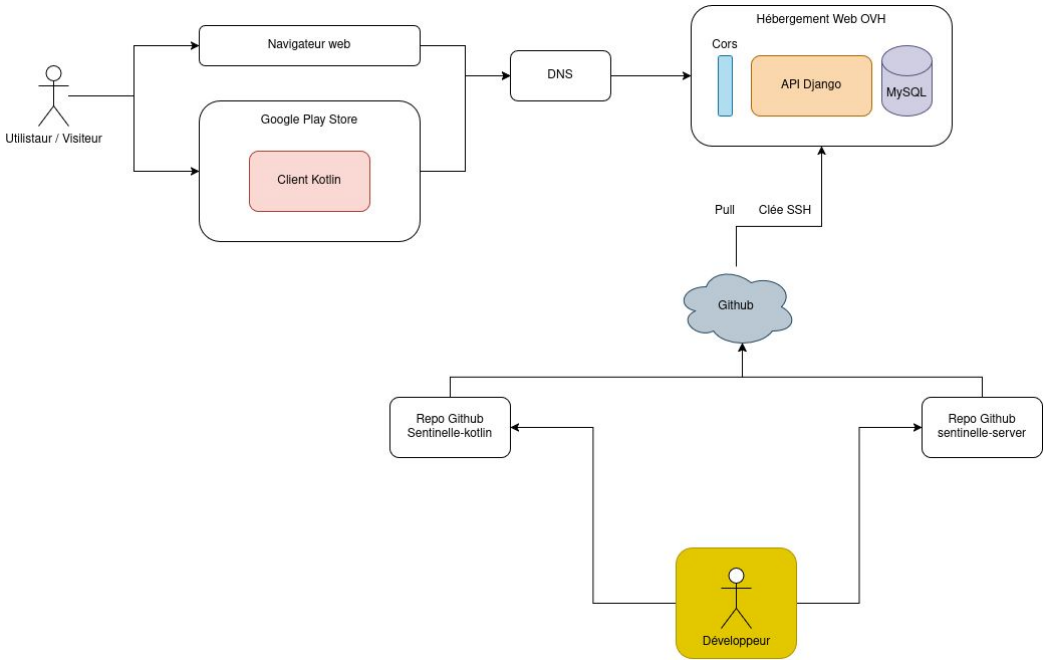
Nom logique : Status

Rubriques

Id	Nom conceptuel	Nom logique	Type	NOT NULL	UNIQUE
🔑	id_status	id_status	INT	✓	✓
	name	name	VARCHAR(50)	✓	

Architecture de déploiement

Voici l'architecture de déploiement de Sentinelle. Elle explique la disposition des composants du système sur les infrastructures physiques et les outils utilisés pour la mise en production.



Serveur Web

Pour le déploiement de l'API, j'utiliserai l'hébergeur OVH pour héberger l'API Django. En plus de cela, j'ajouterai le nom de domaine « sentinelle-app.net », ainsi que le certificat SSL ; le coût annuel est de 94,90 € TTC.

Architecture Logicielle

Les différentes couches seront réalisées dans des dossiers distincts. La partie Api sera intitulée « api », la partie couche applicative pour l'application Android s'intitulera « App » et la partie mobile « app ».

Les deux seront dans un dossier nommé « Sentinelle »

Interface de programmation application web

Les sources et les versions du projet sont gérées par Git. L'application est construite

selon le pattern MVC avec le framework Django. La couche modèle donne l'accès aux données dans la couche de données. Les contrôleurs comprendront toutes les méthodes qui permettent de réaliser les différentes fonctions du CRUD. Les routes serviront à associer des méthodes à des urls.

Couche de présentation

Les sources et les versions du projet sont également générées par Git. J'utiliserai l'architecture par défaut générée par Django. Ces fichiers seront faits en HTML, CSS et JS.

Sécurité et Authentification

L'authentification des utilisateurs ne se fera pas directement avec l'API. Pour ce projet, j'utiliserai le service Auth de Firebase. Le client recevra un token généré par le SDK Firebase et devra le transmettre à l'API, qui interrogera Firebase pour vérifier la validité de ce dernier. Aucun mot de passe ne sera stocké en base de données.

Informations supplémentaires

Environnement

Pour mener à bien ce projet, l'environnement choisi est le suivant :

- L'IDE pour le client kotlin : Android Studio
- L'IDE pour l'API et le client web : Pycharm
- Gestion des versions : Git relié à un compte Github
- Os en local : Ubuntu

Procédure de livraison

Pour livrer le projet, l'API et l'application web seront déployées par un GitHub Action sur le serveur OVH. Le code source sera donc stocké dans un repository. Ainsi, lorsqu'un commit sera effectué et sera mergé sur la branche main, les modifications se déploierons automatiquement sur le serveur.

Le client Kotlin sera également stocké dans un repository, mais sera déployé manuellement sur le Google Play Store. En effet il n'existe pas de pipeline automatique car il est nécessaire de compiler le projet, avec une signature

numérique et de remplir un formulaire pour chaque mise à jours.

DOSSIER PLANS DE TEST

TITRE PROFESSIONNEL

CONCEPTEUR DÉVELOPPEUR D'APPLICATION

Mars 2025 - Mars 2026

Axel Vincent Jaunet

Sommaire

Introduction

Description du projet

Ce projet a pour but de protéger ses utilisateurs en prévenant ses proches en cas de problème. Sentinelle permet donc de se rassurer sans en ne prévenant les proches qu'en cas de réel problème.

Objectifs des tests

L'objectif des tests est de vérifier que tous les composants fonctionne indépendamment l'un de l'autre, de vérifier que l'interface communique bien avec la couche métier et de valider les modifications avant de déployer sur la production.

Inventaire des tests

Voici la liste des tests qui devrons être réalisé pour valider le projet avant de le déployer

Tests unitaires

- TU-001 : Tester l'ajout d'un contact avec un bon et un mauvais format de numéro de téléphone
- TU-002 : Tester la génération d'un path unique d'un Saferider
- TU-003 : Tester le calcul de la date de fin théorique d'un Saferider
- TU-004 : Tester la vérification des conditions de démarrage d'un Saferider
- TU-005 : Tester la détermination du statut lors de l'arrêt d'un Saferider
- TU-006 : Tester la validation des coordonnées GPS
- TU-007 : Tester la logique de vérification de l'accès au fichier
- TU-008 : Tester la génération du fichier GPX
- TU-009 : Tester la conversion du timestamp au format iso

Tests d'intégration

- TI-001 : Tester l'authentification Firebase et création d'utilisateur

- TI-002 : Tester l'écriture et la lecture de coordonnées GPS
- TI-003 : Tester l'upload et le téléchargement de fichier audio
- TI-004 : Tester l'interaction entre les tags et un Saferider
- TI-005 : Tester la suppression en cascade des données lors de la suppression d'un compte
- TI-006 : Tester la structure de fichier lors de la création d'un Saferider
- TI-007 : Tester la gestion d'accès à un contenu non autorisé
- TI-008 : Test de la modification du message
- TI-009 : Test de la suppression de contact avec une simulation d'erreur dans la réponse

Tests fonctionnelles

- TF-001 : Tester le cycle complet de la gestion d'un contact
- TF-002 : Tester le workflow complet d'un Saferider
- TF-003 : Tester le cycle complet de la gestion d'un tag
- TF-004 : Tester la récupération complète des infos d'un Saferider
- TF-005 : Tester la modification d'un profil utilisateur
- TF-006 : Tentative de démarrage d'un Saferider sans les prérequis
- TF-007 : Tester la suppression d'un Saferider

Stratégie de tests

Les tests seront effectués à la fin de la phase de réalisation de l'application. Les tests unitaires seront d'abord réalisés, puis les tests fonctionnels, et pour finir, les tests d'intégration. L'outil par défaut de Django « TestCase » sera utilisé pour effectuer tous les tests de l'API.

Les tests se feront à partir d'un terminal de commande, dans le dossier api de l'application. Tous les tests seront regroupés dans le fichier tests.py.

Tout les test seront lancés avec la commande python3 manage.py test

Tests Unitaires

Cas TU-001

Description : Tester la validation de numéro de téléphone avec un format valide et d'autre aux format invalide

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Définition d'un numéro de téléphone valide	Numéro enregistrer en variable	OK
2	Définition d'une liste de numéro de téléphone invalide	Numéro non validé	OK
3	**Observer le résultat**	**OK (2 tests, 7 assertions)**	**Pass**

Cas TU-002

Description : Tester la génération d'un path unique lors de la création d'un saferider

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Génération de deux path	Paths enregistrer dans deux variables	OK
2	Tester si les deux paths sont identiques	Différents l'un de l'autre	OK
3	**Observer le résultat**	**OK (1 tests, 2 assertions)**	**Pass**

Cas TU-003

Description : Tester la logique de calcule de la date de fin théorique d'un minuteur d'un Saferider

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Récupération du datetime actuel	Timestamp stocker en variable	OK
2	Ajouter 3600 secondes aux timestamp	Résultat stocker dans une variable	OK
3	Comparer le timestamp à la dernière variable de calcule	Timestamp inférieur a l'addition	OK
4	**Observer le résultat**	**OK (1 tests, 2 assertions)**	**Pass**

Cas TU-004

Description : Tester la vérification que les conditions de démarrages sois rempli

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Vérification des variables si remplis (firstname, lastname, phone)	False	OK
2	Vérifier l'existence de contact	False	OK
3	Ajouter un contact	Stocker en variable	OK
4	Vérifier l'existence de contact	True	OK
5	**Observer le résultat**	**OK (2 tests, 3 assertions)**	**Pass**

Cas TU-005

Description : Tester la détermination du statue lors de l'arrêt du minuteur

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Calcule de la date de fin théorique	Résultat stocker en variable	OK
2	Ajouter 3600 secondes aux timestamp	Résultat stocker dans une variable	OK
3	Comparer le timestamp actuel et la date de fin théorique	Détermination d'un status	OK
4	**Observer le résultat**	**OK (1 tests, 4 assertions)**	**Pass**

Cas TU-006

Description : Tester de la validation du format des coordonnées GPS.

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Déclaration des coordonnées valides	Résultat stocker en variable	OK
2	Déclaration des coordonnées invalides	Résultat stocker en variable	OK
3	Appliquer les règles aux coordonnées	True pour les valides et False pour les invalides	OK
4	**Observer le résultat**	**OK (2 tests, 8 assertions)**	**Pass**

Cas TU-007

Description : Vérification de l'accès aux fichiers d'un Saferider

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un Saferider	Résultat stocker en variable	OK
2	Crée un utilisateur	Résultat stocker en variable	OK
3	Vérification de l'accès au Saferider avec le deuxième utilisateur	Accès non autorisé	OK
4	**Observer le résultat**	**OK (1 tests, 2 assertions)**	**Pass**

Cas TU-008

Description : Vérification que la génération du GPX soit correcte

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un Saferider	Résultat stocker en variable	OK
2	Définition de fausses coordonnées	Résultat stocker en variable	OK
3	Génération du GPX	Résultat stocker en variable	OK
4	**Observer le résultat**	**OK (1 tests, 4 assertions)**	**Pass**

Cas TU-009

Description : Vérification de la conversion du timestamp au format iso

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Définition d'un timestamp	Résultat stocker en variable	OK
2	Conversion du timestamp en format iso	Résultat stocker en variable	OK
4	**Observer le résultat**	**OK (1 tests, 4 assertions)**	**Pass**

Tests fonctionnelles

Cas TF-001

Description : Tester le cycle complet de l'ajout d'un contact, le sélectionner et le supprimé

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur
- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création du contact	Code 200	OK
2	Sélectionné le contact	Code 200	OK
3	Supprimé le contact et vérifier que l'objet n'existe plus	Code 200 et nombre d'objet à 0	OK
4	**Observer le résultat**	**OK (3 tests, 6 assertions)**	**Pass**

Cas TF-002

Description : Tester le workflow complet, démarrage du timer, envoi de position, arrêt du timer

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur
- Avoir crée et sélectionné un contact
- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Démarrage du timer	Code 200	OK
2	Envoi des coordonnées GPS	Code 200	OK
3	Arrêt du timer	Code 200 et champs real_end_date égale au timestamp actuel	OK
4	**Observer le résultat**	**OK (3 tests, 4 assertions)**	**Pass**

Cas TF-003

Description : Tester le workflow complet, démarrage du timer, envoi de position, arrêt du timer

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur
- Avoir crée et sélectionné un contact
- Avoir une couleur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Démarrage d'un timer	Code 200	OK
2	Créer un tag	Code 200	OK
3	Associé le tag à un saferider	Code 200	OK
4	Dissocié le tag	Code 200	OK

5	Supprimé le tag	Code 200	OK
6	**Observer le résultat**	**OK (3 tests, 4 assertions)**	**Pass**

Cas TF-004

Description : Tester la récupération des données d'un saferider

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur
- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Démarrage du timer	Code 200	OK
2	Envoi des coordonnées GPS	Code 200	OK
3	Arrêt du timer	Code 200	OK
4	Récupération des information du timer	Code 200	OK
5	**Observer le résultat**	**OK (1 tests, 4 assertions)**	**Pass**

Cas TF-005

Description : Mise à jour des données du profile utilisateur

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur
- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Modifié le nom, prénom et le numéro de téléphone	Code 200	OK
2	Modification du message personnalisé	Code 200	OK

3	Récupération des informations utilisateur	Code 200 et donnée en json	OK
4	Comparaison des ancienne donnée et des nouvelles	AssertEquals valide	OK
5	**Observer le résultat**	**OK (2 tests, 6 assertions)**	**Pass**

Cas TF-006

Description : Test du démarrage d'un timer sans les prérequis

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Crée un utilisateur sans numéro de téléphone	Code 200	OK
2	Démarrer le timer sans contact sélectionner	Code 400	OK
3	Ajouter un contact et démarrer le timer sans message personnalisé	Code 400	OK
4	Ajouter un message personnalisé et démarrer le timer sans téléphone	Code 400	OK
5	Ajouter un numéro de téléphone et démarrer le timer sans prénom	Code 400	OK
6	Ajouter un prénom et démarrer le timer sans nom	Code 400	OK
7	**Observer le résultat**	**OK (5 tests, 10 assertions)**	**Pass**

Cas TF-007

Description : Test de la suppression d'un saferider

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir deux records type (csv et audio)

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un contact	Code 200	OK
2	Démarrage d'un saferider	Code 200	OK
3	Vérification que le dossier existe	Path existe	OK
4	Supprimé le saferider	Code 200	OK
5	**Observer le résultat**	**OK (5 tests, 10 assertions)**	**Pass**

Cas TF-008

Description : Test de la modification du message

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Exécution de la requête post	Code 200	OK
2	Récupération de la réponse	Code 200	OK
3	Insertion de du message dans une variable global	True	OK
4	**Observer le résultat**	**OK (1 tests, 2 assertions)**	**Pass**

Cas TF-009

Description : Test de la suppression de contact avec une simulation d'erreur dans la réponse

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Exécution de la requête post	Code 400	OK
2	Récupération de la réponse	Résultat stocker en variable	OK
3	**Observer le résultat**	**OK (1 tests, 1 assertions)**	**Pass**

Test d’intégration

Cas TI-001

Description : Test de l’authentification avec firebase et création de l’utilisateur

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Envoi vers un endpoint de vérification du token avec un compte inexistant	Code 200	OK
2	Vérification du nouvelle utilisateur	Code 200	OK
3	**Observer le résultat**	**OK (1 tests, 3 assertions)**	**Pass**

Cas TI-002

Description : Test l’envoi de donnée GPS

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d’un contact	Code 200	OK
2	Démarrage du timer	Code 200	OK
3	Envoi d’un tableau de coordonnées	Code 200	OK
4	Récupération des données du trajet	Code 200 + data en json	OK
5	**Observer le résultat**	**OK (1 tests, 3 assertions)**	**Pass**

Cas TI-003

Description : Test l'envoi de fichier audio

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un contact	Code 200	OK
2	Démarrage du timer	Code 200	OK
3	Envoi d'un fichier audio	Code 200	OK
4	Récupération des données du trajet	Code 200 + data en json	OK
5	**Observer le résultat**	**OK (1 tests, 3 assertions)**	**Pass**

Cas TI-004

Description : Test de l'interaction entre les tags et un Saferiders

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un contact	Code 200	OK
2	Création de plusieurs tags	Code 200	OK
3	Démarrage de deux timers	Code 200	OK
4	Association de deux tags sur un saferider	Code 200	OK
5	**Observer le résultat**	**OK (2 tests, 3 assertions)**	**Pass**

Cas TI-005

Description : Test de la suppressions de toute les données lors de la suppression du compte.

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un contact	Code 200	OK
2	Création de plusieurs tags	Code 200	OK
3	Démarrage d'un timer	Code 200	OK
4	Suppression du compte	Code 200	OK
5	Récupération des données du compte	Code 400	OK
6	**Observer le résultat**	**OK (1 tests, 6 assertions)**	**Pass**

Cas TI-006

Description : Test de la gestion de la structure des fichiers lors du démarrage d'un timer

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un contact	Code 200	OK
2	Démarrage d'un timer	Code 200	OK
3	Vérification de la structure	Dossier et fichier existant	OK
4	Vérification des permissions	True	OK
5	**Observer le résultat**	**OK (2 tests, 4**	**Pass**

		assertions)	

Cas TI-007

Description : Test de la vérification de permission d'accès à un contenu d'un autre utilisateur

Auteur du test	Date	Résultat
Axel Jaunet	14/02/2026	Pass

Prérequis :

- Avoir créé un utilisateur

Etapes	Détails	Résultat attendu	Résultats obtenus
1	Création d'un utilisateur 2	Code 200	OK
2	Démarrage d'un timer avec l'utilisateur 1	Code 200	OK
3	Récupération des données du saferider avec l'utilisateur 2	Code 404	OK
4	**Observer le résultat**	**OK (2 tests, 4 assertions)**	**Pass**