

Projet Turbo Plant

Titre professionnel Concepteur développeur
d'applications Niveau VI

Mars 2025
Onat Rigault



1. Présentation du projet en Français	3
2. Présentation du projet en Anglais	4
3. Liste des compétences du référentiel	5
4. Gestion de projet	7
5. Spécifications fonctionnelles	
1. Expression des besoins	9
2. Principales fonctionnalités	9
3. Le parcours utilisateur	10
4. Wireframes et maquettes	11
6. Spécifications techniques	
1. Choix des technologies	
- Pourquoi Kotlin et Compose Multiplatform	13
- Pourquoi Typescript et Express	13
2. Base de données	
- MCD/MLD	14
- Pourquoi PostgreSQL ?	17
7. Développement	
1. Front-End	
- Structure du projet	18
- Composants d'interface avec Jetpack Compose	20
- Connexion avec le back-end (Ktor et Room)	23
- Tests	25
2. Back-End	
- Structure du projet	27
- Les Contrôleurs	28
- Sécurité	30
- L'ORM	34
- Tests	35
3. CI/CD	37
4. Le déploiement	39
8. Description d'une situation de travail ayant nécessité une recherche	41
9. Evolutions potentielles	44
10. Conclusion	45

1. Présentation du projet en Français

Après sept années de travail dans la restauration, et attiré depuis très jeune par la programmation, j'ai fait le choix en 2023 de me réorienter et de tenter ma chance dans ce domaine.

J'ai donc intégré Ada Tech School qui propose une formation avec un premier module de 9 mois à l'école puis un second en 12 mois d'alternance, visant à acquérir la certification RNCP de niveau 6 « Concepteur et développeur d'applications».

J'ai réalisé mon alternance dans l'entreprise Unico, qui propose des solutions informatiques pour accompagner les professionnels du secteur de la collecte de déchets. Un outil web est proposé pour la planification et la gestion des tournées de collectes, en lien avec un outil mobile utilisé par les conducteurs des camions, leur fournissant un outil de navigation, de reporting des tournées réalisées et d'interfaçage avec les objets dont peut être équipé le camion . C'est sur ce dernier que j'ai eu l'occasion de travailler en suivant la méthodologie agile dans l'organisation de notre travail.

C'est dans ce cadre que j'ai donc réalisé ce projet intitulé **Turbo Plant**.

Heureux propriétaire ou en charge d'un (trop) grand nombre de plantes et tristement endeuillé de la perte d'une partie d'entre elles par méconnaissance ou oubli, j'ai décidé d'y remédier avec ce qui me passionne et qui est un outil adapté à ce type de problématique : l'outil informatique.

C'est aussi dans la réflexion de ce que peut être sur le long terme un outil mobile allant vers la domotique dans la gestion de lieux partagés que s'inscrit la démarche de ce projet.

Dans son état d'avancement actuel **Turbo Plant** se veut être une application qui permet de lister ses plantes, de les visualiser, et de gérer des tâches qui y sont associées, tout en partageant ces informations avec d'autres utilisateurs en charge des plantes dans un même lieu.

2. Présentation du projet en Anglais

After seven years of working in the restaurant industry and having been drawn to programming from a young age, I decided in 2023 to change careers and take my chance in this field.

I joined Ada Tech School, which offers a training program consisting of a nine-month initial module at the school, followed by a twelve-month work-study module, aiming to obtain the RNCP Level 6 certification as a « Concepteur et développeur d'applications » which stands for "Application Designer and Developer".

I completed my work-study at Unico, a company that provides IT solutions to support professionals in the waste collection industry. They offer a web based tool for planning and managing collection routes, combined with a mobile tool used by truck drivers. This mobile tool assists drivers with navigation, reporting realised routes, and interfacing with devices installed in the trucks. I had the opportunity to work on this mobile tool, following agile methodology in our workflow.

This is the context in which I developed my project, **Turbo Plant**.

As a proud owner or caretaker of (too) many plants and having sadly lost some due to lack of knowledge or forgetfulness, I decided to address this issue using what I am passionate about and also what can be a good solution for this kind of problems: software tools.

This project is also part of a larger reflection on the long-term potential of mobile tools integrating with home automation for managing shared spaces.

At its current stage of development, **Turbo Plant** is designed to list plants, display information about them, and manage related tasks, while also allowing users to share plant care responsibilities with others in the same location.

3. Liste des compétences du référentiel

Développer une application sécurisée

- Installer et configurer son environnement de travail en fonction du projet**

L'utilisation d'Android Studio et Webstorm avec Github m'ont permis de mettre en place un environnement de travail efficace. Un guide d'installation est trouvable dans les Readme de chaque projet.

- Développer des interfaces utilisateurs**

L'application utilise le framework Jetpack Compose qui permet, grâce à une architecture séparée en composants réutilisables, d'avoir une interface cohérente entre les différents éléments et permettant l'utilisation des outils d'accessibilité.

- Développer des composants métier**

Des composants front-end et back-end tels que les ViewModel où les contrôleurs appliquent des logiques métiers dans les données qu'ils véhiculent.

- Contribuer à la gestion d'un projet informatique**

L'utilisation de Miro pour la conception et le maquettage, et de Github pour le versioning et le découpage des tâches, ont permis de couvrir la gestion du projet tout au long des différentes phases.

Concevoir et développer une application sécurisée organisée en couches

- Analyser les besoins et maquetter une application**

Les étapes de définition d'un persona, d'un MVP et d'un parcours utilisateur et de celles de création de wireframes puis de maquettes ont permis d'établir un plan clair des objectifs de conception avant la phase développement.

- Définir l'architecture logicielle d'une application**

D'une séparation claire des responsabilités de chaque éléments appuyée par l'utilisation de designs patterns, résulte une architecture claire qui rend le code facile à maintenir et robuste face aux erreurs.

- Concevoir et mettre en place une base de données relationnelle**

La conception d'un MCD puis d'un MLD ont établis les liens entre les différentes entités. Ils donnent une structure des schémas d'une base de données qui ont pu être appliqués grâce à un serveur PostgreSQL.

- **Développer des composants d'accès aux données SQL**

La base de données PostgreSQL a pu être gérée et accédée grâce à l'ORM Drizzle. Avec sa syntaxe proche du SQL il m'a permis d'accéder efficacement et de manière sécurisée aux données depuis les contrôleurs. Il a aussi été l'outil utilisé pour définir physiquement les schémas des données et pour appliquer des migrations faisant ainsi évoluer la base de données tout au long du développement.

Préparer le déploiement d'une application sécurisée

- **Préparer et exécuter les plans de tests d'une application**

La rédaction de tests unitaires, exécutés lors du développement ou lors de pipelines à chaque « push » sur le dépôt distant ont permis de détecter rapidement les erreurs et régressions et de s'assurer d'une certaine fiabilité du code envoyé plus tard en production.

- **Préparer et documenter le déploiement d'une application**

L'application est déployée sur un Raspberry grâce à Docker et Nginx et servie via un protocole HTTPS sécurisé.

- **Contribuer à la mise en production dans une démarche DevOps**

L'automatisation des différentes phases avant et au moment de la mise en production s'est effectuée grâce à une pipeline de CI/CD avec Github Actions. Elle exécute les phases de lint, test, build sur toutes les branches et ajoute une phase de déploiement à chaque commit sur la branche de production.

4. Gestion de projet

Même si ce projet a été réalisé de manière individuelle il a utilisé de multiples outils de gestion de projet collaboratifs. Cela permet de se servir d'outils éprouvés par l'usage, adoptant ainsi de bonnes pratiques de travail et rendant accessible à tous moment le travail avec de nouveaux collaborateurs.

J'ai utilisé dès la conception la plateforme Miro, qui offre de nombreux templates qui m'ont permis de poser mes premières idées sous forme de graphiques et schémas comme par exemple avec la réalisation d'un MCD. J'ai aussi pu y réaliser les premiers wireframe, puis des maquettes finales de l'application.

En concentrant ces réalisations dans le même espace de travail, la plateforme joue le rôle de documentation et de suivi des processus de réflexion autour de la conception de l'application.

Pour ce qui est du **versioning** de mon code j'ai utilisé l'outil **Git** avec la plateforme Github pour l'hébergement distant. Il y a un repository différent pour l'application mobile et pour le back-end. J'ai utilisé des branches différentes pour le développement de chaque fonctionnalités, parce que même dans un travail individuel cela facilite la relecture qui est d'autant plus importante sans possibilité de revue de code.

Exemple d'une « merge request » :

11 - Fix tests in InstantExts and AuthRepository #12

Merged onatyr merged 1 commit into develop from 11-onatyr 4 days ago

Conversation 0 Commits 1 Checks 0 Files changed 2

onatyr commented 4 days ago
No description provided.

Deleted Instant test + fixed AuthRepository tests 8e06be8

onatyr linked an issue 4 days ago that may be closed by this pull request
Fix tests in InstantExts and AuthRepository #11 Closed

onatyr merged commit fb13012 into develop 4 days ago Revert

Pull request successfully merged and closed
You're all set — the branch has been merged.

J'ai aussi utilisé la plateforme Github en ce qui concerne la séparation des tâches et leur suivi grâce aux **Github Issues**. En effet l'utilisation d'un tableau kanban pour faire ça aurait déplacé ce suivi sur un autre outil sans nécessité lié au travail de groupe. Cela permet à n'importe quelle personne qui a accès au repository de voir les issues et les pull request associées. Il est aussi possible de suivre leur état d'avancement, de priorité et d'y ajouter des labels pour mieux les catégoriser.

A screenshot of a GitHub Issues Kanban board. The board has three columns: 'To Do', 'In Progress', and 'Done'. There are six cards visible:

- To Do:** create basic plant detailed card (feature) PRIO: High. Status: Open. Last updated Jan 24. 1 comment.
- In Progress:** inject Resources to use getString from everywhere (non fonctionnal) PRIO: Medium WIP. Status: In Progress. Last updated Jan 24. 1 comment.
- In Progress:** add gradle task to remove database (non fonctionnal) PRIO: Low. Status: In Progress. Last updated Jan 18. 1 comment.
- In Progress:** create an eventbus to display snackbar (enhancement) PRIO: Low. Status: In Progress. Last updated Jan 18. 1 comment.
- In Progress:** add regex on password & email + email verification (enhancement) PRIO: Medium. Status: In Progress. Last updated Jan 18. 1 comment.
- To Do:** create basic screen for rooms/places (feature) PRIO: High. Status: Open. Last updated Jan 24. 1 comment.

Pour m'organiser dans le temps j'ai regroupé, grâce au « Milestones » de Github, la réalisation des tâches en sprints distincts correspondants à des phases du projet comme l'analyse du besoin, la mise en place du repository et des dépendances, le développement d'une grosse fonctionnalité.

Last mockup integrations

⚠ Past due by 23 days 100% complete

	0 Open	4 Closed
<input type="checkbox"/> ⚡ create basic screen for rooms/places (feature) PRIO: High	#3 by onatyr was closed on Jan 24	1 comment
<input type="checkbox"/> ⚡ create basic plant detailed card (feature) PRIO: High	#9 by onatyr was closed on Jan 24	1 comment
<input type="checkbox"/> ⚡ create basic screen for tasks (feature) PRIO: High	#2 by onatyr was closed on Jan 24	1 comment
<input type="checkbox"/> ⚡ create basic screen to add newPlant + plant ID (feature) PRIO: High	#1 by onatyr was closed on Dec 30, 2024	1 comment

5. Spécifications fonctionnelles

5.1. Expression des besoins

Pour réfléchir à la conception autour d'un ensemble de besoins proches et cohérents entre eux j'ai défini un **persona**.

C'est une personne amatrice de plantes sans en être experte qui en a la charge dans un ou plusieurs espaces partagés comme une colocation et un bureau. Elle aimerait pouvoir ajouter et retrouver facilement des informations sur des plantes qu'elle ne connaît pas forcément. Elle a aussi besoin de planifier des tâches d'entretien, comme l'arrosage et d'avoir des rappels à échéance proche et suivre leur état de réalisation. Elle a besoin que ces plantes et les informations et tâches associées soient partagées avec d'autres personnes en fonction de lieux communs.

5.2. Principales fonctionnalités

Ce **persona** permet d'établir une liste de fonctionnalités principales et qui vont constituer un produit minimal viable (MVP).

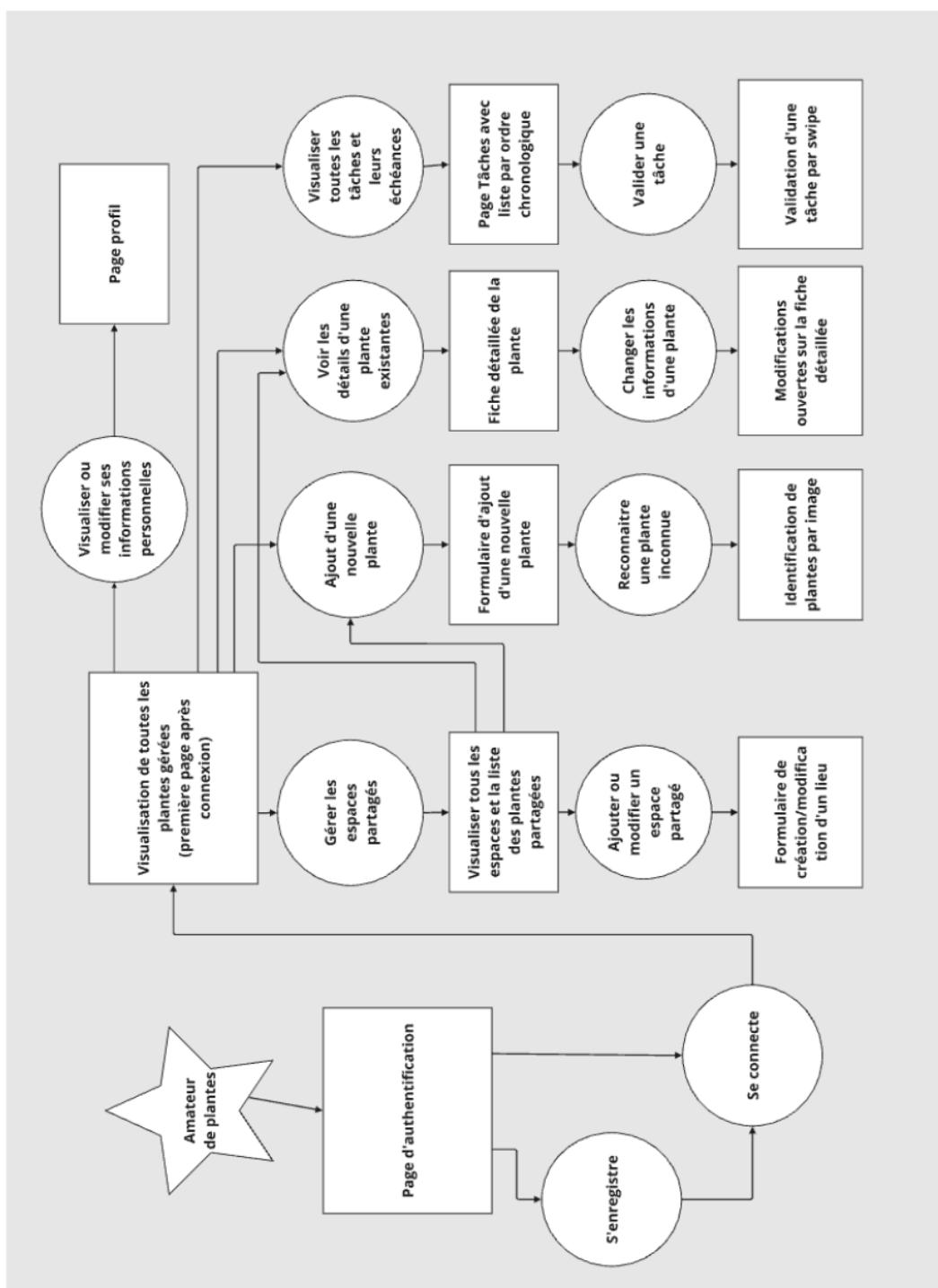
- **Un système d'authentification**, de la création à la connexion d'un compte, avec une page profil pour la gestion du compte.
- **L'ajout d'une nouvelle plante à sa liste** avec possibilité d'y ajouter des informations détaillées, en étant si besoin aidé d'une reconnaissance de plantes par images si on ne connaît pas l'espèce.
- **La possibilité d'y ajouter des tâches ponctuelles ou récurrentes** comme l'arrosage, de les visualiser, de recevoir des rappels et de suivre leur état de réalisation.
- **Le partage des plantes avec d'autres utilisateurs** en fonction de lieux et pièces communes.
- **La visualisation des plantes entretenues** en listes, par lieux ou de manière détaillée.

5.3. Le parcours utilisateur

La définition de la liste des principales fonctionnalités m'a permis d'imaginer un parcours utilisateur. Ce parcours utilisateur représente **le chemin emprunté par mon persona dans l'application**.

Il met en lien les besoins, dans une description étape par étape du parcours qu'offre l'application pour répondre de la manière la plus pertinente à ceux ci.

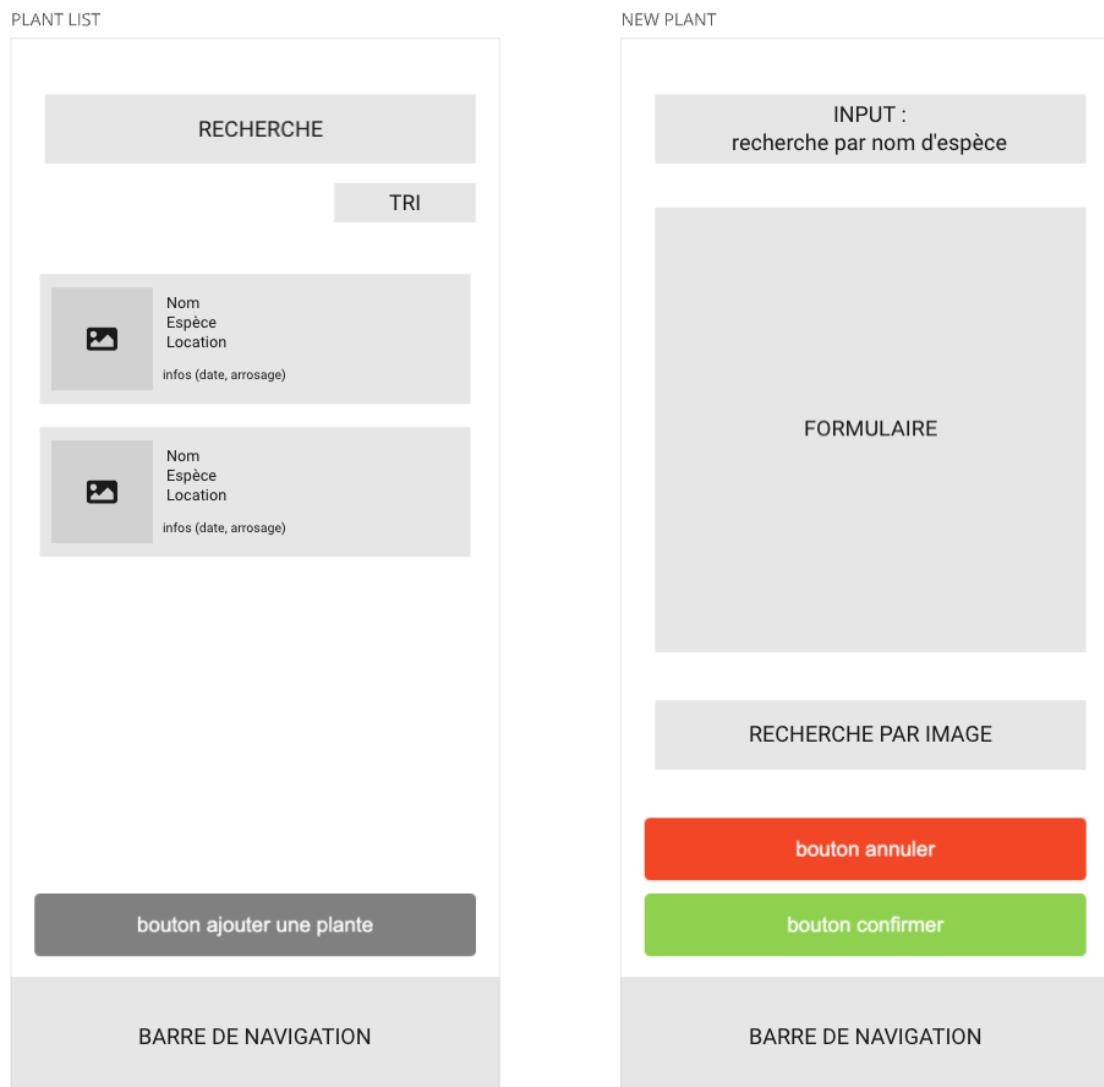
Les cercles représentent les besoins et les rectangles l'étape dans l'application.



5.4. Wireframes et maquettes

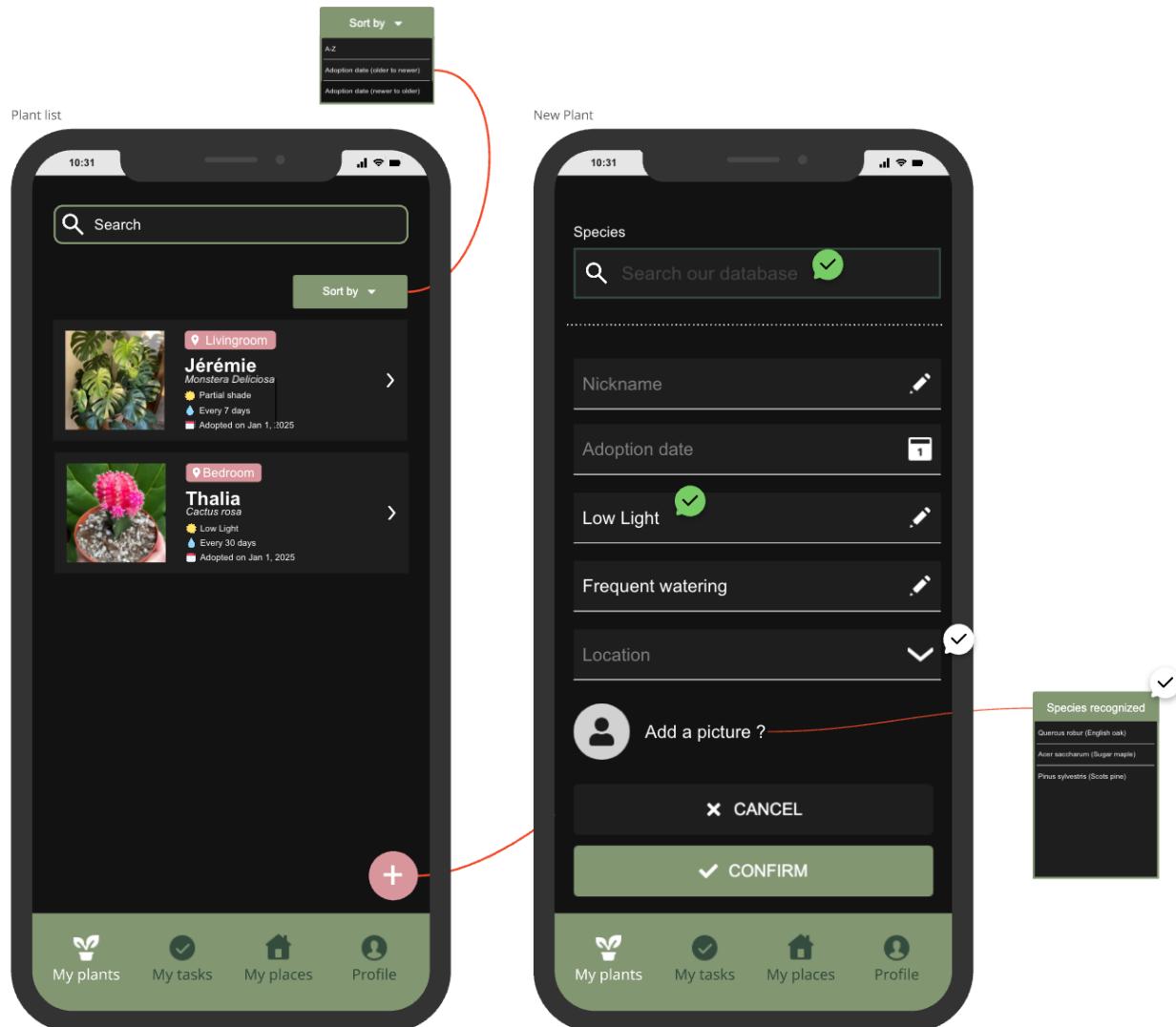
Avec un parcours utilisateur établi j'ai commencé à créer des **wireframes** qui représentent le squelette des différentes pages. J'y ai décrit l'arrangement des différents composants entre eux en intégrant la représentation des éléments de navigation d'une page à une autre, suivant ainsi le parcours établi par le parcours utilisateur.

On a à cette étape le plan d'une application presque complètement fonctionnelle et décrivant l'UX (Expérience utilisateur).



J'ai ensuite « habillé » ces wireframes et créé **les maquettes de l'application**.

Ces maquettes seront par la suite utilisées comme références durant le développement et la création des composants d'interface.



6. Spécifications technique

6.1. Choix des technologies

Pourquoi Kotlin avec Compose Multiplatform (KMP et CMP) ?

Tout d'abord pour le front-end de l'application j'ai choisi de me diriger vers une solution mobile et multiplateforme. En effet les usages du mobile prennent de plus en plus de place avec une augmentation à 50.7 % du trafic internet contre 46.7 % pour le desktop (chiffre en baisse quant à lui). De plus cette solution répond aux besoins d'utilisation en extérieur et permet de tirer avantage de l'appareil photo embarqué pour intégrer plus naturellement la fonctionnalité d'ajout de photo et de reconnaissance de plante.

Technologie émergente ces dernières années, Kotlin et Compose Multiplatform sont maintenant considérées comme prête à l'utilisation en production . Elles permettent de partager les logiques métiers et l'UI, ont des performances plus proches du natif que leur concurrentes React Native et Flutter et me permettent une mise en place plus rapide grâce à mes connaissances déjà acquises en développement Android avec Kotlin.

Pourquoi Express et Typescript ?

En ce qui concerne le back-end mon choix s'est tournée vers l'écosystème Javascript avec son framework back Express codé en Typescript. Kotlin aurait pu être une option, permettant à nouveau de partager du code et d'accélérer le temps de développement mais dans le cas de l'ajout futur d'une application web de nouvelles contraintes se seraient rajoutées. Les interfaces web sont parfois plus riches en fonctionnalités que les interfaces mobiles et les développeurs Javascript en plus grand nombre que ceux initiés à Kotlin. Pour ces raisons le choix de technologies Javascript pour le back-end partagé m'a paru évident. Typescript pour un code plus robuste et moins sensibles aux erreurs, et Express pour son assise en terme d'utilisation qui en fait un outil facile d'adoption avec un support fort.

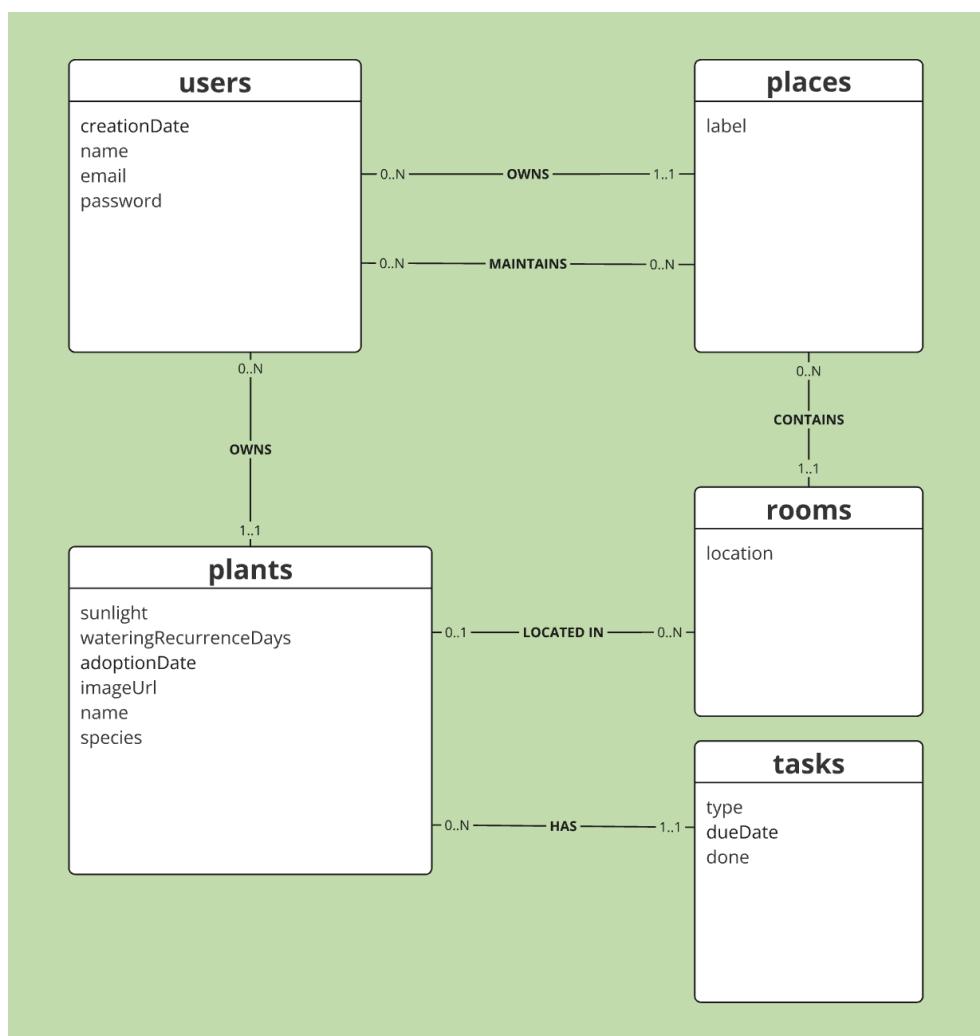
6.2 Bases de données

MCD/MLD

Pour concevoir la structure de ma base de données et définir leur implémentation j'ai utilisé des éléments de la méthodes Merise à savoir le MCD pour « Modèle Conceptuel des Données » dans un premier temps.

Le MCD représente la logique métier de la donnée. Elle les sépare en **entités** distinctes, leur défini des **attributs** et décrit la **nature de leurs relations**. On peut résumer son objectif par définir ce que sont les données et comment elles sont reliées.

Voici donc la modélisation conceptuelle réalisée :



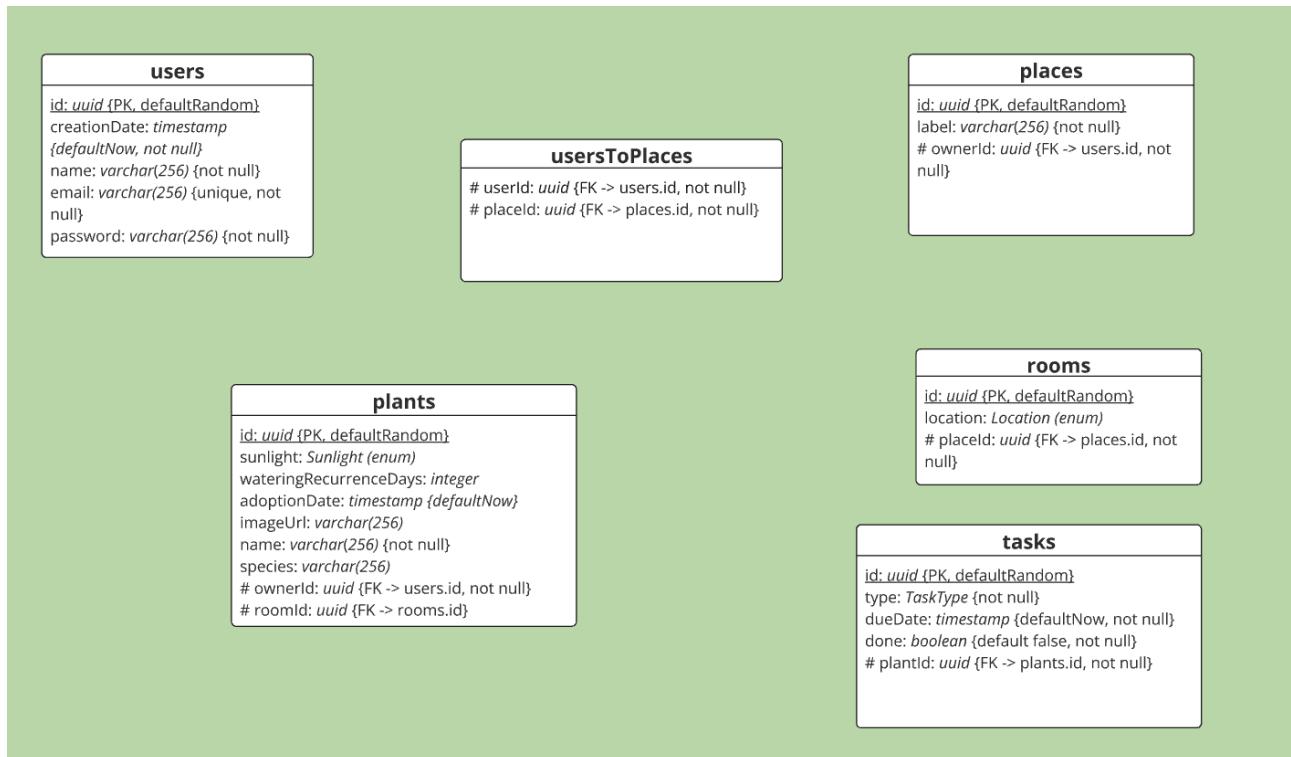
Chaque entité ici possède des attributs qui décrivent sa nature de manière non redondante.

En ce qui concerne les associations, elles sont des liens sémantiques entre les différentes entités. On y ajoute un lien de cardinalité qui représente un nombre

minimum et maximum d'associations entre une ou plusieurs occurrence d'une entité et une ou plusieurs occurrence d'une autre.

Dans le cas de la relation entre « places » et « rooms », on peut déduire de cette représentation qu'une « place » peut contenir une « room » à raison d'aucune ou d'un nombre illimité de « room », qui elle ne peut être contenue qu'au minimum et au maximum une seule fois.

De cette modélisation conceptuelle peut être construite une modélisation logique de ces données, le MLD, qui va nous servir à implémenter notre base de données grâce à ce schéma relationnel.



Les entités sont maintenant des **tables** qui contiennent des champs et les relations entre elles sont représentées par des **clefs étrangères** (FK) correspondant à la **clef primaire** d'une autre table (PK). Par exemple pour représenter l'association d'une tâche à une plante, la table « tasks » contient un champ « plantId » qui est une clef étrangère associée à la clef primaire « id » de la table « plants ».

Chaque champs est représenté par un type de donnée et peut posséder des contraintes d'**unicité** et de **non nullabilité**. Une table intermédiaire apparaît aussi pour représenter l'association de maintenance d'un user et d'une place. En effet un user peut maintenir plusieurs place et une même place peut être maintenue par plusieurs users. On crée donc une autre table, sans clef primaire qui va représenter chaque combinaison user/place.

Cette modélisation va être la référence lors de la création du **schéma de base de données** comme réalisé sur l'exemple qui suit avec l'ORM Drizzle.

```
export const plants : PgTableWithColumns<{ name: 'plants'; sche... = pgTable('plants', { Show usages ± onatyr +1
  id: uuid('id').primaryKey().defaultRandom(),
  userId: uuid('user_id')
    .references(() : PgColumn<{ name: 'id'; tableName: 'users'... => users.id)
    .notNull(),
  species: varchar('species', {length: 256}),
  name: varchar('name', {length: 256}).notNULL(),
  sunlight: sunlightEnum('sunlight'),
  wateringRecurrenceDays: integer('wateringRecurrenceDays'),
  adoptionDate: timestamp('adoption_date').defaultNow(),
  roomId: uuid('room_id').references(() : PgColumn<{ name: 'id'; tableName: 'rooms'... => rooms.id),
  imageUrl: varchar('image_url'),
});
});
```

L'utilisation du MLD avec un ORM permet de générer une implémentation de ce que pourrait être un MPD (Modèle physique de données)

```
CREATE TABLE IF NOT EXISTS "plants" (
  "id" uuid PRIMARY KEY DEFAULT gen_random_uuid() NOT NULL,
  "user_id" uuid NOT NULL,
  "species" varchar(256),
  "name" varchar(256) NOT NULL,
  "sunlight" "sunlight",
  "wateringRecurrenceDays" integer,
  "adoption_date" timestamp DEFAULT now(),
  "room_id" uuid,
  "image_url" varchar
```

```
ALTER TABLE "plants" ADD CONSTRAINT "plants_user_id_users_id_fk" FOREIGN KEY ("user_id") REFERENCES "public"."users"("id") ON DELETE no action ON UPDATE no action;
ALTER TABLE "plants" ADD CONSTRAINT "plants_room_id_rooms_id_fk" FOREIGN KEY ("room_id") REFERENCES "public"."rooms"("id") ON DELETE no action ON UPDATE no action;
```

Pourquoi PostgreSQL ?

Pour mettre en place ma base de données j'ai choisi PostgreSQL qui est un SGBDR (Système de Gestion de Base de Données Relationnelles). En effet la structure des données conçue avec le MCD et le MLD nécessite la mise en place d'une base de données relationnelles.

PostgreSQL renforce les références à des clefs étrangères en s'assurant de leur existence lors de leur création et peut empêcher la suppression ou mettre à jour les données lorsqu'un champ d'une table référencée est modifiée, s'assurant ainsi de l'intégrité des données.

PostgreSQL optimise aussi les requêtes comportant des jointures permettant d'écrire ainsi des requêtes complexes sans impacter les performances.

Ce système de gestion est par ailleurs open source, facilement extensible et supporte nativement le type JSONB, apportant ainsi certains avantages du NoSQL et nous permettant de stocker si besoin des données non structurées.

Tout cela en fait donc un outil adéquat pour créer et requérir les entités conçues et permet aussi une adaptabilité si de nouveaux besoins apparaissent (données non structurées, extensions pour compléter les fonctionnalités).

7. Développement

7.1. Front-End

Structure du projet

Le design pattern **MVVM (Model-View-ViewModel)** est la norme dans le développement mobile.

Il permet de maintenir séparées différentes couches de l'application. La partie « Model » représente la donnée et va être implantée principalement par les classes « Repository » qui vont se charger des requêtes dans la base de données locale ou celles distantes en exposant des méthodes pour interagir avec ces données. On fera généralement un Repository par entité et ses méthodes s'apparenteront à un CRUD.

La partie « View » représente la couche de présentation, c'est à dire d'UI/UX et regroupe les différents écrans et composants que comporte l'interface utilisateur. Ces deux parties communiquent grâce au « ViewModel » implémentée par la classe du même nom qui se chargera de faire la jonction entre les interactions sur l'interface et les états de la données, en servant les données à la vue et en les modifiant par le biais des Repository. Il y a généralement un ViewModel par écran différent et ses méthodes, dans leur logique et leur naming peuvent retranscrire les logiques métiers.

En termes d'architecture, une application Compose Multiplatform se découpe en trois dossier principaux, commonMain pour le code partagé, et iosMain et androidMain qui reprennent la même architecture mais qui ne comportent que les fichiers qui contiennent du code spécifique à chaque plateforme, reflétant les différences d'implémentations lorsqu'il y en a.

Fonction dans commonMain/.../libs/utils/ComposableUtils

```
expect fun getScreenSize(): ScreenSize
```

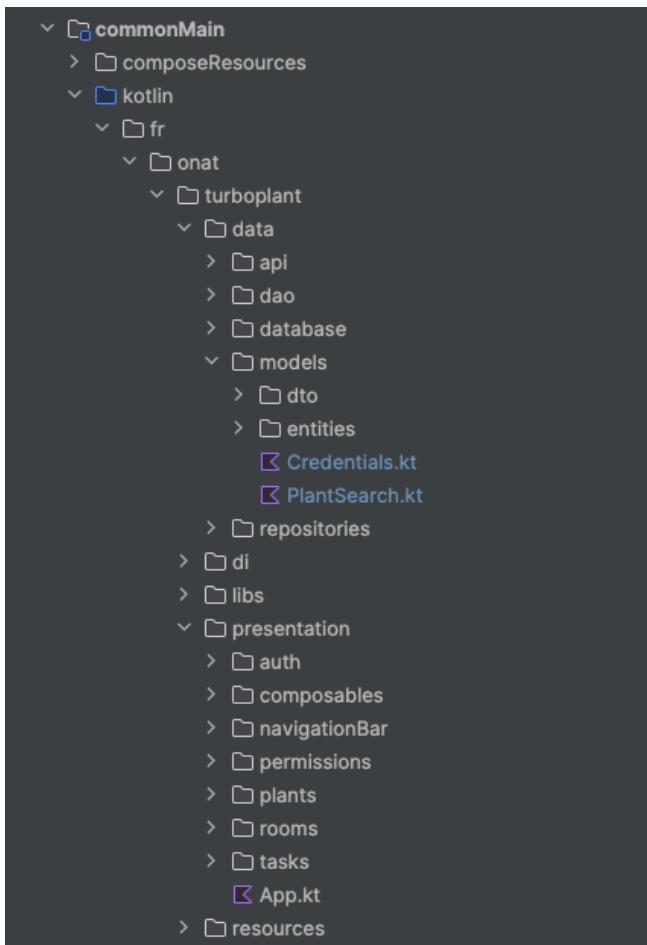
Fonction dans iosMain/.../libs/utils/ComposableUtils

```
actual fun getScreenSize(): ScreenSize = ScreenSize(
    widthDp = LocalWindowInfo.current.containerSize.width.pxToPoint(),
    heightDp = LocalWindowInfo.current.containerSize.height.pxToPoint()
)
```

Fonction dans androidMain/.../libs/utils/ComposableUtils

```
actual fun getScreenSize(): ScreenSize = ScreenSize(
    widthDp = LocalConfiguration.current.screenWidthDp,
    heightDp = LocalConfiguration.current.screenHeightDp
)
```

Le dossier commonMain quant à lui se compose principalement entre **la couche de données** et **la couche de présentation**,



aux cotés de ressources pour les assets, de libs pour les fonctions utilitaires et de di pour l'injection de dépendances.

Composants d'interface avec Jetpack Compose

Pour réaliser les éléments de l'interface j'ai utilisé la librairie Jetpack Compose, basée sur les principes de programmation réactives. Similaire dans le web à React, on utilise des composants qui observent des valeurs et se recréée en fonction des changements d'état.

Dans la réalisation j'ai suivi les normes en créant bien un « Screen » différent pour chaque vue. Chacun d'entre eux a son ViewModel associé duquel il collecte les valeurs observées qui sont repassées aux composants enfants.

Ils sont comme les chefs d'orchestre de la composition et vont contenir des logiques parfois plus complexes que leurs enfants desquels ont va essayer de limiter la responsabilité.

```
@Composable
fun AuthScreen(
    viewModel: AuthViewModel = koinViewModel(),
    navigate: () -> Unit
) {
    val signMode by viewModel.signMode.collectAsStateWithLifecycle()
    val loginDetails by viewModel.loginDetails.collectAsStateWithLifecycle()
    val registrationDetails by viewModel.registrationDetails.collectAsStateWithLifecycle()

    val scope = rememberCoroutineScope()
    val snackBarHostState = LocalSnackbarHostState.current
    val showSnackBarMessage: (String?) -> Unit = {
        scope.launch {
            snackBarHostState.showSnackbar(message: it ?: "")
        }
    }

    viewModel.isAuthenticated.collectAsEffect { if (it) navigate() }

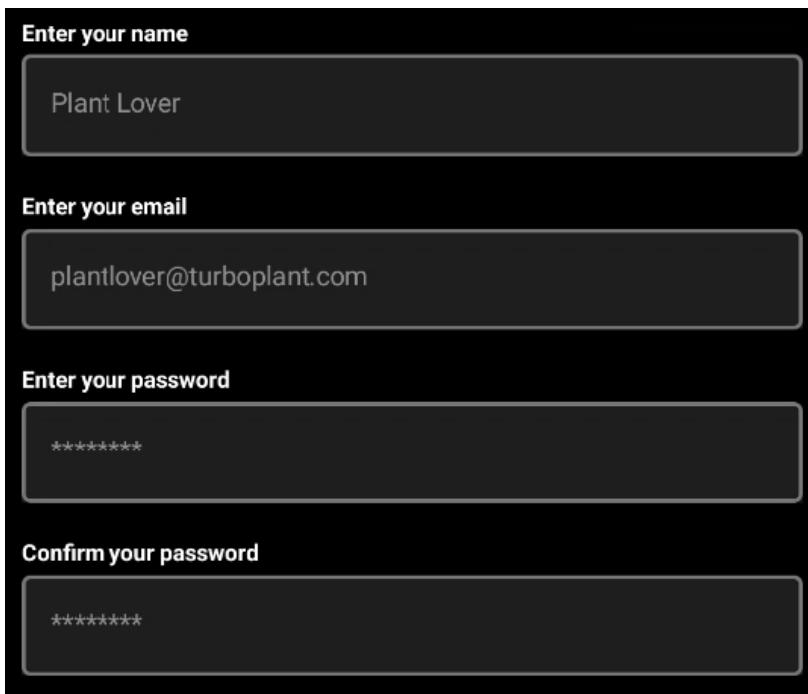
    Column(
        modifier = Modifier
            .background(Color.Black)
            .fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Top
    ) {
        AuthHeader()
        SignModeToggle(currentSignMode = signMode, updateSignMode = viewModel::updateSignMode)
        if (signMode == SignMode.IN)
            LoginForm(
                loginDetails = loginDetails,
                updateLoginDetails = viewModel::updateLoginDetails,
                validateForm = { viewModel.sendLoginRequest(showSnackBarMessage) }
            )
        else
            RegistrationForm(
                registrationDetails = registrationDetails,
                updateRegistrationDetails = viewModel::updateRegistrationDetails,
                validateForm = { viewModel.sendRegistrationRequest(showSnackBarMessage) }
            )
    }
}
```

Au plus bas dans la hiérarchie on va retrouver des composants de bases réutilisés à plusieurs endroits comme le « BaseTextField ». Cette réutilisation permet évidemment de simplifier le code pour ne pas se répéter mais aussi de garder une continuité dans le design de chaque élément.

```
@Composable
fun NameField(
    value: String,
    imeAction: ImeAction = ImeAction.Next,
    updateValue: (String) -> Unit
) {
    BaseTextField(
        labelRes = Res.string.enter_your_name,
        placeHolderRes = Res.string.name_placeholder,
        keyboardOptions = KeyboardOptions(
            keyboardType = KeyboardType.Text,
            imeAction = imeAction,
            autoCorrectEnabled = true
        ),
        value = value,
        updateValue = updateValue
    )
}

@Composable
fun EmailField(
    value: String,
    imeAction: ImeAction = ImeAction.Next,
    updateValue: (String) -> Unit
) {
    BaseTextField(
        labelRes = Res.string.enter_your_email,
        placeHolderRes = Res.string.email_placeholder,
        keyboardOptions = KeyboardOptions(
            capitalization = KeyboardCapitalization.None,
            keyboardType = KeyboardType.Email,
            imeAction = imeAction,
            autoCorrectEnabled = false
        ),
        value = value,
        updateValue = updateValue
    )
}
```

Utilisation du BaseTextField dans les formulaires d'authentification :



Parfois aussi certains éléments, comme le «LoginForm» ou «RegistrationForm» qui comportent ces différents Fields, ne sont pas réutilisé mais l'encapsulation dans un composant à part entière permet une meilleure compréhension du rôle de chacun et une meilleure lisibilité.

La navigation entre les différentes vues de l'application est coordonnées depuis la racine de l'arbre de composants. C'est le rôle du NavHost qui se comporte comme un routeur, où la liste des différentes routes y est enregistrée et on peut naviguer d'un écran à un autre grâce à la méthode « navigate » du « NavController ».

```
NavHost(  
    navController = navController,  
    startDestination = LoginRoute,  
    modifier = Modifier.weight(1f)  
) {  
    composable<LoginRoute> {  
        AuthScreen(navigate = {  
            navController.navigate(PlantsRoute) {  
                navController.graph.startDestinationRoute?.let { startDestination ->  
                    popUpTo(startDestination) {  
                        inclusive = true  
                    }  
                }  
                launchSingleTop = true  
            }  
        })  
    }  
    composable<PlantsRoute> { PlantListScreen(navigate = { navController.navigate(it) }) }  
    composable<AddNewPlantRoute> {  
        NewPlantScreen(navigate = { navController.navigate(it) })  
    }  
    composable<CameraRoute> {  
        PlantIdentificationScreen(Modifier.fillMaxSize())  
    }  
    composable<TasksRoute> { TasksScreen() }  
    composable<RoomsRoute> { RoomsScreen() }  
}
```

Connexion avec le back-end

Pour la connexion avec le back-end j'ai utilisé la librairie Ktor comme **client HTTP** car éprouvée et aussi utilisée côté serveur dans des projets en Kotlin et aussi pour sa compatibilité avec un projet multiplateforme.

Le client est ensuite utilisé dans la classe ArchiApi qui expose des méthodes de CRUD classiques et simplifiée pour leur utilisation.

```
class ArchiApi(
    private val client: IHttpClient,
    private val userDao: UserDao,
) {

    private val baseUrl = "https://api.turboplant.onat.fr"

    suspend fun get(
        routeUrl: String,
    ): HttpResponse? {
        val token = userDao.getToken().first()
        try {
            val url = "$baseUrl$routeUrl"
            return client.get {
                url { url(url) }
                token?.let { headers.append(name: "Authorization", value: "Bearer $it") }

            }
        } catch (e: Exception) {
            logger(e.message)
        }
        return null
    }
}
```

Tous les appels à l'API passent donc par cette classe et sont réalisés dans les différents Repository.

Ils récupèrent la donnée distante et la stockent dans une base de données locale ce qui permet une utilisation partielle de l'application hors connexion.

Pour la persistance des données localement j'ai utilisé la librairie Room. Elle se base sur SQLite et permet de rédiger des requêtes en **SQL** et d'exposer des méthodes via un DAO (Data access object) pour appeler ces requêtes agissant ainsi comme un ORM.

```
@Dao
interface PlantDao {
    @Upsert
    suspend fun upsert(plant: Plant)

    @Upsert
    suspend fun upsertAll(plants: List<Plant>)

    @Query("DELETE FROM Plant")
    suspend fun clear()

    @Query("SELECT * FROM Plant")
    fun getAllWithRoom(): Flow<List<PlantWithRoom>>

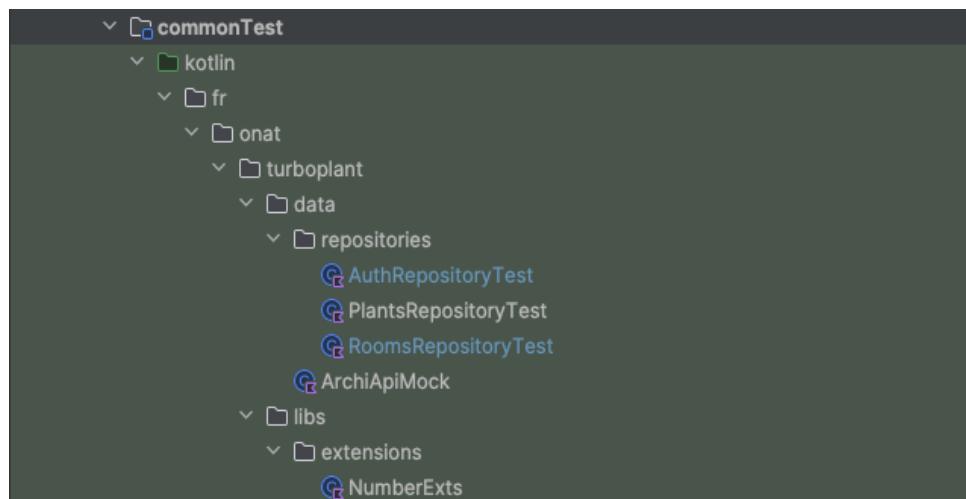
    @Query(
        """SELECT * FROM Plant
        WHERE roomId = :id"""
    )
    fun getPlantsByRoomId(id: String): Flow<List<Plant>>
}
```

Tests

Pour améliorer le processus de développement et réduire le risque d'erreurs dans mon code j'y ai intégré des tests, qui pourront être lancés durant le développement au besoin et systématiquement lors des push et merges sur les branches Git et lors du déploiement.

Pour l'application mobile j'ai utilisé la librairie native de Kotlin, Kotlin Tests avec l'aide de la librairie Mockative pour générer les mocks de certaines classes. Ces mocks sont une implémentation factices de la réelle implémentation dans le sens où ses méthodes n'exécutent pas de réel code et on pourra définir le comportement de ces fonctions à l'exécution du code pour leur faire avoir le comportement souhaité.

Les tests de commonMain se trouvent dans le dossier commonTest situé dans le même dossier que ce premier et suivent une architecture similaire à celle des fichiers qu'on va vouloir tester.



Les tests dans commonTest peuvent être exécuté sur le système de chaque plateforme séparément pour vérifier que les résultats correspondent à ce qui est attendu indépendamment des environnements spécifiques.

Dans l'image suivant on peut voir qu'on utilise des mocks comme décrit précédemment sur la classe ArchiApi et UserDao.

En effet dans les tests de l'AuthRepository on ne veut pas dépendre des résultats de l'exécution des méthodes de ces deux premières classes mais plutôt avoir la possibilité de choisir leur comportement pour définir un cas d'usage et tester le comportement isolé de nos fonctions, selon ces comportements pré-définis.

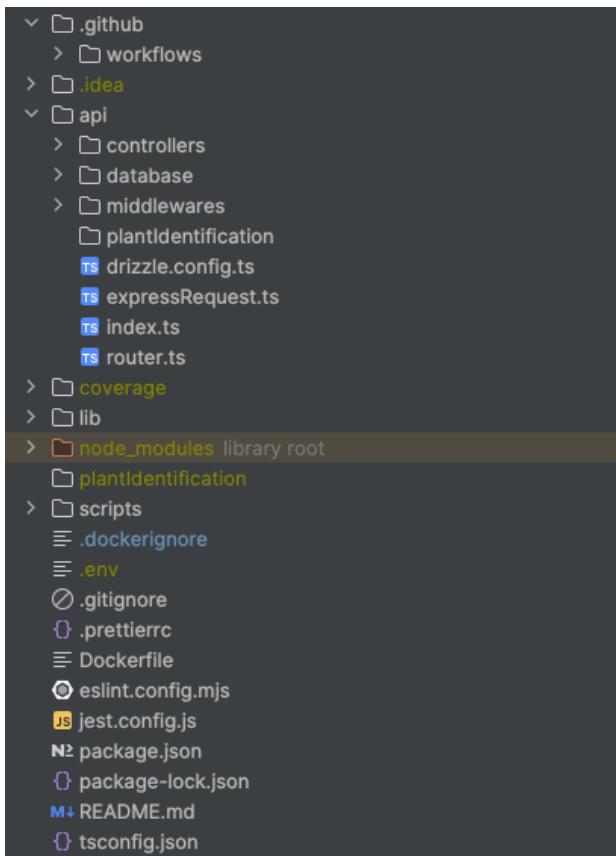
Chaque test dans son nommage va décrire le comportement attendu par la fonction testée dans un contexte, et suivre une structure dite de « **GIVEN, WHEN, THEN** » (« ETANT DONNÉ», « QUAND », « ALORS »).

En prenant l'exemple du premier test on peut donc lire : ETANT DONNÉ que userDao.getToken() retourne un Flow contenant une valeur « mockToken », QUAND on attribue la valeur renournée par authRepository.isAuthenticated() à la variable isAuthenticated, ALORS la première valeur contenue dans isAuthenticated est un booleéan de valeur « true »,

```
18 > class AuthRepositoryTest {
19
20     private val archiApi = mock(of<ArchiApi>())
21     private val userDao = mock(of<UserDao>())
22     private val authRepository = AuthRepository(archiApi, userDao)
23
24     @Test
25     fun `isAuthenticated returns true when a token is returned`() {
26         every { userDao.getToken() }.invokes { flowOf( value: "mockToken" ) }
27
28         val isAuthenticated = authRepository.isAuthenticated()
29
30         runBlocking {
31             assertTrue(isAuthenticated.first())
32         }
33     }
34
35     @Test
36     fun `isAuthenticated returns false when no token is returned`() {
37         every { userDao.getToken() }.invokes { flowOf( value: null ) }
38
39         val isAuthenticated = authRepository.isAuthenticated()
40
41         runBlocking {
42             assertFalse(isAuthenticated.first())
43         }
44     }
}
```

7.2. Back-End

Structure du projet



En ce qui concerne l'architecture du back end on va d'abord retrouver les éléments spécifique à ce type de projet dans le dossier « api », il contient le fichier index.ts, exécuté pour faire tourner le server, le routeur qui va rediriger les requêtes en fonction du contenu de l'URL vers les contrôleurs adaptés.

Ces requêtes passent par les middlewares comme celui d'authentification qui vérifie que chaque requête contient un token valide. Et enfin les contrôleurs qui contiennent la logique appelée par chaque « endpoint ».

Dans ce dossier on retrouve aussi le dossier database et le fichier drizzle.config qui concerne les composants d'accès à la base de données.

A la racine on retrouve des éléments plus communs à n'importe quel projet comme des fichiers de configuration, comme par exemple pour le lint (eslint.config), les tests (jest.config) et Typescript (tsconfig).

On retrouve aussi le Dockerfile et le dockerignore pour générer des images Docker, les package.json et nodes modules pour la gestion de dépendance et enfin le dossier .github pour la CI/CD et le gitignore pour lister les fichiers/dossiers qu'on ne veut pas garder dans le versioning.

Dans ces fichiers dont on a besoin dans le projet mais qu'on veut garder hors du versioning on peut trouver par exemple le fichier .env qui contient les variables d'environnement qu'on a besoin de garder secrètes par mesure de sécurité.

Les contrôleurs

Ils sont les chainons finaux du traitement d'une requête HTTP. C'est la fonction qu'ils exécutent qu'on cherche à lancer lorsqu'on fait un appel sur un endpoint.

La nature des actions qu'ils vont effectuer peut être déduite de la méthode utilisée pour faire la requête. En effet, étant les composants interagissant avec la base de données ce sont eux qui constituent ce qu'on appelle le **CRUD**. C'est pour cette raison qu'on va regrouper les fonctions des contrôleurs par entité.

L'exemple suivant illustre une fonction du tasksController qui gère l'entité « tasks ». On peut identifier plusieurs éléments la constitue. Ligne 24 : le nom du contrôleur, la méthode utilisée, le nom du endpoint et les éléments de la requête passées en paramètres.

On en comprend que cette fonction sert à mettre à jour le statut d'une task à complétée.

```
24  tasksController.put('/complete/:id', async (req, res, next) => {
25    try {
26      const taskId = req.params.id;
27      const [completedTask] = await db
28        .update(tasks)
29        .set({ done: true })
30        .where(eq(tasks.id, taskId))
31        .returning({ plantId: tasks.plantId, type: tasks.type });
32
33      if (!completedTask) {
34        return res.status(500).json({ message: 'Failed to complete the task' });
35      }
36
37      const [plant] = await getPlantById(completedTask.plantId).execute();
38      if (plant.wateringRecurrenceDays) {
39        await insertNewTaskTask(
40          plant.id,
41          'watering',
42          computeNextOccurrence(plant.wateringRecurrenceDays)
43        );
44      }
45
46      res.status(200).json({ message: 'Task completed' });
47    } catch (e) {
48      console.error(e);
49      res.status(500).json({ message: 'Internal Server Error' });
50    }
51  });
```

La fonction complète la tâche en exécutant une requête à la base de données (ligne 27 à 31), puis s'il n'y a pas d'erreur (ligne 33 à 35) et que la plante possède une valeur dans le champ « wateringRecurrenceDays », rajoute une nouvelle tâche à l'occurrence suivante (ligne 37 à 44).

On peut voir plusieurs fois lors de l'exécution des appels à « res.status() ». Cela permet d'ajouter un message à ce qui va constituer la réponse à la requête et qui sera retourné au client. On peut y indiquer comme par exemple ici, si la fonction a été exécutée correctement ou interrompre le fil d'exécution et retourner une erreur.

La responsabilité du contrôleur est donc principalement de gérer la **couche applicative** en s'occupant de la requête HTTP. Au vu de la complexité de mon projet j'ai laissé dans le contrôleur de la logique métier mais elle pourrait à l'avenir être séparée dans un « Service » qui encapsulerait cette partie.

Sécurité

Sur un projet full-stack les questions de sécurité sont en grande partie la responsabilité du back-end. En effet. Le back-end a un accès direct aux données ce qui en fait un composant particulièrement sensible aux failles de vulnérabilité.

J'ai mis en places quelques pratiques élémentaires qui permettent de réduire ces risques.

L'une d'entre elles est un système d'authentification. On veut s'assurer que l'utilisateur est bien celui qu'il dit être, que ses accès n'ont pas été compromis et qu'il n'a accès qu'aux données pour lesquelles il est autorisé.

On commence donc avec la partie de création de compte et le stockage des mots de passe. Pour empêcher qu'un mot de passe soit divulgué en cas de fuites de données on va utiliser un algorithme de **hashage**. La fonction va transformer le mot de passe en entrée en ce qu'on appelle un hash et c'est ce dernier qu'on va sauvegarder.

L'opération de hashage est non réversible sans brute force, il est donc impossible de retrouver le mot de passe d'origine depuis le hash. J'utilises la librairie **bcrypt** pour réaliser cette opération au moment de la création du compte pour ne sauvegarder que le hash, puis à chaque tentative de connexion pour comparer le hash du mot de passe rentré avec celui enregistré en base de données.

```

authController.post(
  '/register',
  async (req: express.Request<unknown, unknown, User>, res) => {
    try {
      const existingUser = await db
        .select()
        .from(users)
        .where(or(eq(users.email, req.body.email), eq(users.name, req.body.name)))
        .limit(1)
        .execute();

      if (existingUser.length > 0) {
        return res.status(400).json({message: 'Name or email already in use'});
      }

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

      const result = await db
        .insert(users)
        .values({
          name: req.body.name,
          email: req.body.email,
          password: hashedPassword,
        })
        .execute();

      return res.status(201).json({message: 'User created successfully'});
    } catch (error) {
      console.error(error);
      return res.status(500).json({message: 'Registration failed'});
    }
  }
);

```

Lors de la connexion, si les deux hash correspondent le back-end renverra un token d'authentification. C'est une chaîne de caractère qui contient des informations comme un id, un nom d'utilisateur et une date d'expiration. Cette chaîne de caractère est signée grâce à une clé secrète ce qui la rend infalsifiable sans cette même clef. Pour effectuer cette partie là j'utilise la librairie **JsonWebToken** qui se charge de générer et signer le token.

Ce token est donc renvoyé à chaque requête postérieure et c'est toujours la même librairie qui s'occupe de le vérifier pour le valider.

Pour ce qui est de la vérification du token j'utilise un **middleware d'authentification**. Ce middleware est appelé à chaque fois qu'une requête arrive, et avant de la passer au contrôleur.

Il va vérifier plusieurs choses : Si l'endpoint ciblé nécessite ou pas une authentification, s'il en nécessite une, la présence et la validité du token. Si la requête est considéré comme autorisée, elle est passée aux contrôleurs on y ajoutant en paramètres de la requête l'id user et le username. Dans le cas contraire elle bloque la requête en renvoie une erreur 401 correspondant au statut « Unauthorized ».

```
export function authenticate(
  req: express.Request,
  res: express.Response,
  next: NextFunction
) : void | Response<any, Record<string, any>> {
  try {
    if (isExempted(req.path)) return next();
    const token : string | undefined = req.headers.authorization?.split(' ')[1];
    if (!token || !process.env.ACCESS_TOKEN_SECRET)
      return res.status(401).json();

    verify(
      token,
      process.env.ACCESS_TOKEN_SECRET,
      (err : VerifyErrors | null, userPayload: JwtPayload | string | undefined) : void => {
        const user = userPayload as JwtUserModel;
        console.log(user);
        req.userId = user.id_user;
        req.username = user.username;
        next();
      }
    );
  } catch (e) {
    console.error(e);
  }
}

export function isExempted(url: string): boolean {
  for (const exemptedEndpoint of EXEMPTED_ENDPOINTS)
    if (url.includes(exemptedEndpoint)) return true;
  return false;
}

const EXEMPTED_ENDPOINTS : string[] = [
  'auth/login',
  'auth/register',
  'plants/identify',
  // endpoints below this message should only be there for testing purpose
];
```

Un autre principe à appliquer tout au long du développement est celui de ne **jamais faire confiance au client**. Cela passe par exemple par faire attention aux messages renvoyées : lors d'une connexion échouée il est préférable de ne pas dire si c'est l'adresse mail ou le mot de passe qui est incorrecte pour ne pas divulguer qu'un compte existe dans le cas où c'est le mot de passe qui est erroné. On risque de donner des informations à d'éventuels attaquants ce qui constituerait une vulnérabilité.

On doit aussi toujours vérifier, au delà de l'authentification, que les demandes sont légitimes.

Par exemple, grâce au contrôleur suivant on peut récupérer toutes les rooms associées à une place grâce à l'id de la place uniquement. Si on ne vérifiait pas que l'utilisateur qui fait la requête est mainteneur de la place, il suffirait d'avoir l'identifiant de la place pour en récupérer les informations.

```
placesController.get('/allRoomsByPlaceId/:placeId', async (req, res, next) => {
  try {
    const hasAccess = !(await getAllPlacesByUserId(req.userId).execute()).find(
      (place) => place.id === req.params.placeId
    );

    if (!hasAccess) {
      res.status(403).json({message: 'No access'});
      return;
    }
    const allRooms = await db
      .select()
      .from('rooms')
      .where(eq('rooms.placeId', req.params.placeId))
      .execute();

    res.status(200).json(allRooms);
  } catch (e) {
    res.status(500).json({message: e});
  }
});
```

Un dernier élément et sûrement l'un des plus critiques est **l'ORM (Object-Relational Mapper)**.

C'est un outil qui permet de manipuler une base de données à travers des objets plutôt que des requêtes SQL. Il simplifie l'accès aux données, améliore la sécurité (en évitant les injections SQL) et facilite la maintenance en rendant les interactions avec le reste du code plus intuitives.

L'ORM

L'ORM est l'outil principal pour manipuler la base de données de manière sécurisée. Mon choix s'est tourné vers Drizzle principalement pour sa proximité syntaxique avec le SQL.

```
const result = await db
  .select()
  .from(users)
  .where(eq(users.email, req.body.email))
  .limit(1)
  .execute();
```

Il m'a permis aussi de gérer les schéma et d'appliquer les migrations lors de changement sur ce dernier.

Le schéma décrit la structure des tables de la base de donnée comme leurs champs, leur types ou leurs relations. Les tables du schéma correspondent à des objets Javascript qu'on peut insérer dans les requêtes en réduisant le risque d'erreur.

Les fichiers de migrations peuvent être générés grâce à la commande **npx drizzle-kit generate** et appliqués grâce à la commande **npx drizzle-kit migrate**.

Le fichier de migration généré est écrit en SQL et est intégré au versioning du projet. On peut ainsi l'appliquer sur la base de données principale lors d'un merge ou un historique des modifications qui ont été apportées, permettant de faire un retour en arrière en cas de problèmes.

Schéma d'une table avec Drizzle :

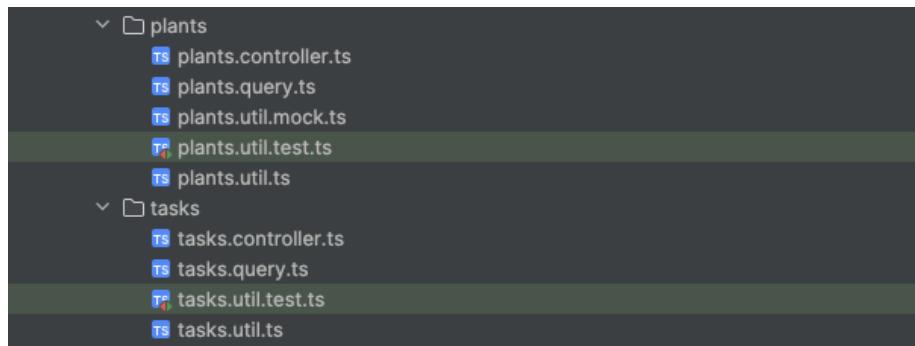
```
export const plants = pgTable('plants', {
  id: uuid('id').primaryKey().defaultRandom(),
  userId: uuid('user_id')
    .references(() => users.id)
    .notNull(),
  species: varchar('species', {length: 256}),
  name: varchar('name', {length: 256}).notNull(),
  sunlight: sunlightEnum('sunlight'),
  wateringRecurrenceDays: integer('wateringRecurrenceDays'),
  adoptionDate: timestamp('adoption_date').defaultNow(),
  roomId: uuid('room_id').references(() => rooms.id),
  imageUrl: varchar('image_url'),
});
```

Tests

De la même manière que pour l'application mobile j'ai intégré des tests unitaires sur mon back-end en utilisant la librairie Jest.

Ces tests unitaires sont écrits pour les fonctions utilitaires dont la nature se prête à ce type de tests. Ils sont des unités isolée avec des valeurs d'entrée et des valeurs de sortie.

Les fichiers de test sont situés à cotés des fichiers testés et comportent le suffixe .test précédé du nom du fichier testé. Ils sont eux aussi exécutés lors des push et merges sur les branches distantes et peuvent aussi être exécutées lors du développement grâce à la commande « npm run test ».



Comme pour les tests unitaires de l'application mobile, ils suivent la structure de « GIVEN, WHEN, THEN »

Dans l'exemple suivant le GIVEN correspond aux mocks d'input, le WHEN à l'appel de la fonction formatPlantsWithTasks avec son input mocké, le THEN à l'**assertion** d'égalité avec le résultat attendu.

```
describe( name: 'formatPlantsWithTasks', fn: () : void => { + onatyr
  test( name: 'should format properly plants with no task in input', fn: () : void => {
    expect( actual: formatPlantsWithTasks( plantsWithTask: mocks.plantWithNoTaskInput)).toEqual(
      expected: mocks.plantWithNoTaskExpected
    );
  });

  test( name: 'should format properly plants with single task in input', fn: () : void => {
    expect( actual: formatPlantsWithTasks( plantsWithTask: mocks.plantWithSingleTaskInput)).toEqual(
      expected: mocks.plantWithSingleTaskExpected
    );
  });

  test( name: 'should format properly plants with multiples tasks in input', fn: () : void => {
    expect( actual: formatPlantsWithTasks( plantsWithTask: mocks.plantWithMultipleTasksInput)).toEqual(
      expected: mocks.plantWithMultipleTasksExpected
    );
  });

  test( name: 'should format properly plants with tasks is duplicated in input', fn: () : void => {
    expect( actual: formatPlantsWithTasks( plantsWithTask: mocks.plantWithTaskDuplicatedInput)).toEqual(
      expected: mocks.plantWithTaskDuplicatedExpected
    );
  });
});
```

J'ai rédigé plusieurs tests pour une même fonction en essayant de couvrir un maximum de cas représentatifs des valeurs d'entrée possibles.

En décrivant tous les comportements attendus de manière exhaustive, les tests peuvent servir de documentation de la fonction et permettent d'en changer l'implémentation à tout moment tout en s'assurant qu'on ne crée pas de régression.

7.3. CI/CD

La CI/CD pour « **continuous integration** » et « **continuous delivery** » est l'ensemble des procédés qui permettent d'automatiser les phases de build, de validation (lint/test) et de déploiement. Elles se lancent lors de différents évènements comme un push sur une branche, ou en faisant un merge sur une branche principale. On peut détecter des erreurs plus tôt et de manière systématique, tout en livrant le produit le plus vite possible.

Ces processus sont décrits dans des fichiers yaml qui détaillent quand et comment chaque phase s'exécute.

```
deploy:
  if: github.ref == 'refs/heads/prod'
  runs-on: ubuntu-latest
  needs: docker-build

  steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Download Docker Image Artifact
      uses: actions/download-artifact@v4
      with:
        name: archi-back-image-${{ github.run_id }}
        path: ./

    - name: Copy Image to Remote Server
      uses: appleboy/scp-action@v0.1.7
      with:
        host: ${{ secrets.RSPB_HOST }}
        username: ${{ secrets.RSPB_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        passphrase: ${{ secrets.SSH_PASSPHRASE }}
        source: ./archi-back.tar
        target: /tmp
        port: ${{ secrets.RSPB_PORT }}

    - name: Load and Run the Docker Image on Server
      uses: appleboy/ssh-action@v1.2.1
      with:
        host: ${{ secrets.RSPB_HOST }}
        username: ${{ secrets.RSPB_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        passphrase: ${{ secrets.SSH_PASSPHRASE }}
        port: ${{ secrets.RSPB_PORT }}
        script: |
          export RSPB_USER=${{ secrets.RSPB_USER }}
          docker ps -a --filter "name=archi-back" -q | grep -q .
          && docker rm -f archi-back \
          && docker rmi -f archi-back:latest \
          || echo "Container archi-back doesn't exist"
          docker load < /tmp/archi-back.tar
          sleep 15
          docker run -d --name archi-back --env-file /home/$RSPB_USER/.env -p 3000:3000 archi-back:latest
```

Dans mon cas j'ai défini les phases de lint, de test et de build sur tous les push de n'importe quelle branche et ai rajouté une phase de déploiement sur les pull request vers la branche de production.

Pour le déploiement j'ai fais le choix d'utiliser Docker pour contenir l'application back-end. Docker permet d'isoler le fonctionnement de l'application dans un environnement indépendant des spécificités de la machine sur laquelle le code s'exécute et des dépendances dont notre application a besoin. On s'assure ainsi qu'un même Dockerfile soit capable de générer des images de notre projet compatibles sur n'importe quelle machine.

Une fois qu'une image est générée la pipeline de déploiement copie cette image puis on la lance dans un container Docker sur le serveur de production.

7.4. Le déploiement

Pour déployer mon application, j'ai fait le choix d'utiliser un Raspberry Pi pour plusieurs raisons. Tout d'abord, cela me permet de gérer moi-même tous les aspects de l'administration d'un serveur. Ensuite, il y a une dimension économique, car le Raspberry est une solution abordable. Enfin, dans l'idée de pouvoir, à terme, partager des plantes sur le même réseau local, cela me semblait être une bonne option.

J'ai utilisé OVH Cloud pour rediriger les connexions de mon domaine vers mon serveur.

Pour accéder au serveur, j'ai d'abord configuré mon routeur en redirigeant le port externe du routeur vers le port interne du Raspberry Pi, et lui attribué une adresse IP fixe, ce qui permet de le rendre accessible depuis l'extérieur. Pour une communication en HTTP, la connexion s'effectue sur le port 80, tandis qu'en HTTPS, cela se fait sur le port 443.

Toutes les connexions au serveur se font via une connexion sécurisée SSH. Ce tunnel permet d'exécuter des commandes à distance, et c'est grâce à cela que j'ai pu administrer le serveur.

Sur le serveur, c'est Nginx qui se charge de rediriger les connexions vers mon back-end, de manière sécurisée. Quant à la gestion de la certification pour établir la connexion HTTPS, elle est prise en charge par Certbot.

Configuration Nginx :

```
server {
    server_name api.turboplant.onat.fr;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/api.turboplant.onat.fr/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/api.turboplant.onat.fr/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

server {
    if ($host = api.turboplant.onat.fr) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen 80;
    listen [::]:80;
    server_name api.turboplant.onat.fr;
    return 404; # managed by Certbot
}
```

Pour le déploiement de l'application mobile j'ai utilisé Fdroid qui est une alternative open source au Googe Play store.

Fdroid permet d'héberger l'application soit dans le dépôt Fdroid soit de créer et d'héberger soi même un dépôt que les utilisateurs pourront accéder et qui est servi de la même manière que le back-end via Nginx à l'adresse **fdroid.turboplant.onat.fr**.

A chaque nouvelle version de l'application l'APK est téléchargée dans le dépôt via une connexion SSH et la commande « fdroid update -c » est exécutée dans ce dépôt et se chargera de vérifier la clé de signature de l'application et créera les métadonnées nécessaires pour servir l'application dans un client Fdroid.

L'application est déployée automatiquement dans la pipeline de CI/CD mais les étapes à suivre peuvent être retrouvées dans le Readme de l'application pour un déploiement manuel.

Deployment

The app is deployed automatically on each new push on the **prod** branch. However if you need to deploy it manually these are the steps:

1. Add the following variables in the local.properties file with the right values

```
KEY_ALIAS=key-alias
KEY_PASSWORD=key-password
STORE_FILE_PATH=/Users/Me/StoreFilePath
STORE_PASSWORD=store-password
```

2. Upgrade the version code and the version name in the app level build.gradle file

```
defaultConfig {
    applicationId = "fr.onat.turboplant"
    minSdk = libs.versions.android.minSdk.get().toInt()
    targetSdk = libs.versions.android.targetSdk.get().toInt()
    versionCode = 2
    versionName = "1.1"
}
```

3. Build the app with the following command

```
./gradlew assembleRelease
```

4. Append the version name to the file name and transfer it on the distant server at the following path

```
/usr/share/nginx/www/fdroid/repo
```

5. Update the metadata by executing the following command at the same path on the distant server

```
fdroid update -c
```

8. Description d'une situation de travail ayant nécessité une recherche

Lors du déploiement de l'application mobile précédemment décrit j'ai fais face à une erreur.

En voici la « stack trace » affichée dans la console :

Cette stack trace représente la pile d'exécution du programme au moment de l'erreur. Elle permet de comprendre ce contexte d'exécution et affiche à la dernière ligne l'erreur renvoyée par le programme. On peut lire « androguard.core.bytecodes.axyml.ResParserError : res1 must be zero ! »

On comprend que l'erreur est lié au programme androguard utilisé par fdroid pour analyser l'application pour appliquer des mesures de sécurité.

J'ai donc copié-collé l'erreur sur mon moteur de recherche qui m'a retourné les résultats suivants :

The screenshot shows a search results page with four entries:

- GitHub** <https://github.com/androguard/androguard/issues/1014> :
ResParserError: res1 must be zero! · Issue #1014 · androguard ...
It seems that the typeSpec struct as defined in the main here still states that both res0 and res1 must be zero. So it appears it is not something coming from updates in the Android source, though it is evident that several apps now create resources with res0 and res1 havin...
This is fixed in latest master branch, but not available in the docker image... "res0 must be always zero!" wh... androguard.core.bytecodes.axml.ResP res0 must be always zero! System...
- GitHub** <https://github.com/MobSF/Mobile-Security-Framework-MobSF/issues/2352> :
ResParserError: res1 must be zero! #2352 - GitHub
This is fixed in latest master branch, but not available in the docker image yet. The docker builds are failing due to pyQT dependency issue for ARM arch. You could build an image locally or use the code from latest master to fix this.
- GitHub** <https://github.com/androguard/androguard/issues/771> :
"res0 must be always zero!" when reading an APK #771 - GitHub
androguard.core.bytecodes.axml.ResParserError: res0 must be always zero! I have seen that bug several times recently too. "res0 must be always zero!" when reading an APK #927. I'm seeing this with com.whatsapp Version Code 230905004, Version 2.23.9.5, SHA-256...
- GitLab** <https://gitlab.com/fdroid/fdroidserver/-/issues/1192> :
res0 must be always zero (#1192) · Issues · F-Droid ... - GitLab
Traceback (most recent call last): File "/usr/lib/python3/dist-packages/androguard/core/bytecodes/apk.py", line 1556, in get_android_resources return self.arsc["resources.arsc"] KeyError: 'resources.arsc' During handling of the above exception...

Le premier lien affiché fait référence au repository Github d'Androguard et notamment à ses Github Issues, fonctionnalité qui permet à la communauté d'utilisateurs de reporter des erreurs rencontrées où les équipes de développement ou d'autres membres de cette communauté peuvent répondre si une solution existe ou mettre la résolution dans les tâches à faire.

C'est donc généralement une très bonne source lorsqu'on a un problème spécifique à un programme.

Dans le premier lien j'ai donc pu y lire que la gestion de l'erreur avait été mal codée et aurait du seulement l'afficher dans la console sans interrompre l'exécution.

Un membre de l'équipe de développement a mis un lien vers une merge request résolvant le problème et recommande de mettre à jour le package vers une nouvelle version.

Ce package étant installé comme dépendance de fdroid je ne pouvais pas la mettre à jour avec le gestionnaire de package « pip » pour les packages python mais j'ai du utiliser le gestionnaire « apt » sur le raspberry.

La mise à jour n'ayant pas été trouvée et n'étant pas sur de la manière de mettre à jour dans ce contexte là je suis retourné sur mes résultats de recherches et suis allé sur le lien 4^e lien qui menait aux Gitlab Issues du repository de Fdroid décrivant la même erreur. Je m'assurais ainsi de trouver des réponses plus proches de mon cas d'utilisation.

J'y ai trouvé cette solution expliquant comment mettre à jour androguard avec le gestionnaire apt et j'ai pu résoudre mon problème et exécuter ma commande sans interruption du programme.

delthas @delthas · 1 month ago

I fixed this on my end by installing the [sid package for androguard](#):

```
curl -L -O http://ftp.us.debian.org/debian/pool/main/a/androguard/androguard_3.4.0~a1-16_all.deb  
sudo apt install ./androguard_3.4.0~a1-16_all.deb
```

9. Evolutions potentielles

En cherchant à répondre d'abord à un MVP, on s'assure de pouvoir mettre dans les mains d'éventuels utilisateurs une version qui répond de **manière simple aux besoins les plus importants**, permettant des **retours d'expérience le plus tôt possible** et ainsi d'adapter la direction que va prendre le développement en restant toujours le plus proche de ce qui est réellement nécessaire.

On peut imaginer aujourd'hui plusieurs fonctionnalités à venir dans la continuité du MVP :

- Une amélioration des informations sur la plante en fonction de son espèce
- Des recommandations sur la base des informations recueillies
- Le partage de la gestion d'une ou plusieurs plantes en dehors des lieux en communs
- L'ajout de nouveaux types de tâches comme le rempotage, taillage ou la fertilisation
- Une adaptation des différentes tâches selon des conditions changeantes (température, humidité) basées sur des données recueillies via des capteurs ou des API de météo.
- La gestion de plantes via un réseau internet local.

Ces pistes de développement devront faire l'objet d'un découpage en plus petites tâches permettant un **développement itératif** et de tests réalisés par des utilisateurs pour une **adaptation continue aux besoins réels**.

10. Conclusion

Turboplant m'a permis de d'approfondir des aspects du développement applicatif comme celui de la conception, qui donne à étudier les raisons de la création d'une application en évaluant **la pertinence et la priorité des besoins**, et de la **structuration des objectifs fonctionnels** grâce à des modélisations de composants de données ou d'interface qui permettront de répondre ainsi à ces objectifs.

J'y ai aussi découvert une nouvelle technologie avec Kotlin Multiplatform, un outil qui m'a permis de surmonter certaines contraintes liées à l'implémentation technique des besoins exprimés lors de la conception, et d'en comprendre les limites. Il m'a aussi permis d'approfondir des sujets que j'avais déjà abordé en essayant d'en comprendre tous les tenants et aboutissants.

Tout ça dans un temps long tout en étant contrainte par la disponibilité, où une organisation rigoureuse et une bonne gestion globale du projet et de ses priorités a été nécessaire pour respecter les objectifs de l'exercice.

Je termine ce projet en ayant une meilleure compréhension globale de tous les enjeux dans lesquels s'inscrit le développement d'une application, de ses contraintes, des limites aussi de certaines méthodes ou outils de travail mais surtout de leur efficacité quand on les applique tôt et de manière systématique.

Enfin, je ressors de cette expérience avec un sentiment renouvelé de mon intérêt pour ce domaine et des défis qui sont à y relever.