Université Gustave Eiffel

Master Informatique

Rapport technique

Android : Plutus

Jimmy Teillard - Irwin Madet - Lorris Creantor - Cédric Chevreuil - Frédéric Xu

Alternants

Chargé de Cours : Michel Chilowicz

M2

Mars 2023

Table des matières

Table des figures

1 Introduction

Plutus est une application de gestion de finances. En effet, un utilisateur est capable de créer un carnet de comptes (aussi appelé "Book"). Chaque carnet de compte possède des informations et données qui lui sont propres : Des transactions (et toutes les informations qui leur sont liées), des étiquettes (tags), et des filtres sauvegardés.

Ce rapport à pour but de détailler et décrire les différents aspects de l'application et de son développement.

2 Structure du projet

Le projet est divisé en trois parties principales, dont deux liées directement à l'application mobile.

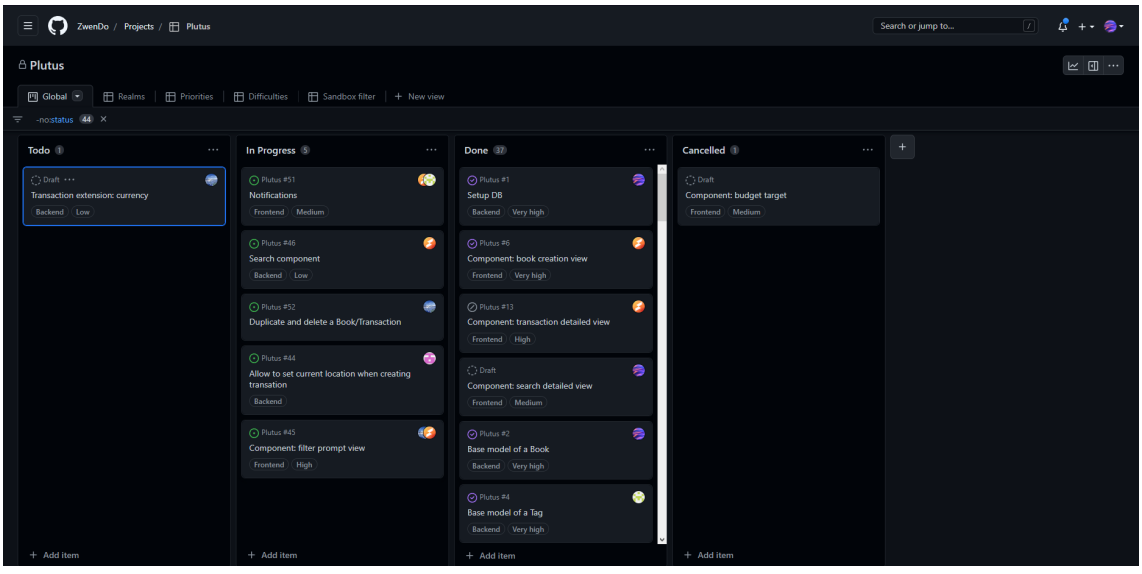
La première partie, située dans `/app/src/main/java/fr/uge/plutus/backend`, concerne, comme son nom l'indique, le backend de l'application mobile. En effet, c'est ici que le modèle de la base de donnée est décrit, ainsi que toutes les fonctionnalités de sérialisation, export, import, etc.

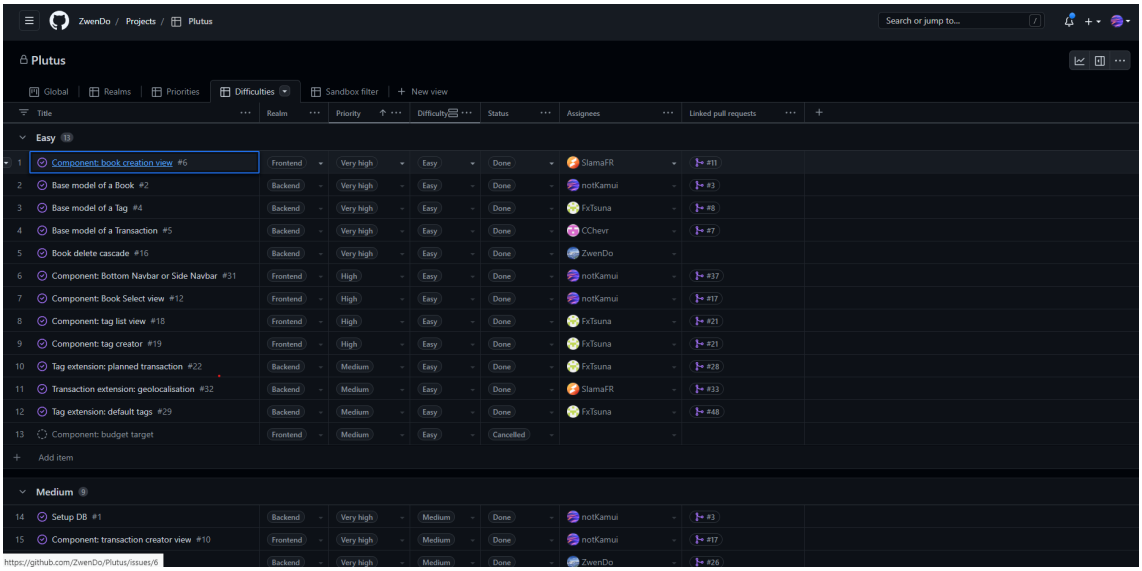
La deuxième partie, située dans `/app/src/main/java/fr/uge/plutus/frontend`, concerne la vue de l'application en elle même, c'est-à-dire l'activité ainsi que tous ses composants.

La troisième et dernière partie, située dans `/service` est un module Gradle qui est une application Ktor et est un microservice permettant la réception de fichiers Plutus et leur stockage, en plus de leur restitution grâce à des JWT uniques.

3 Organisation

Nous nous sommes très rapidement organisés grâce à une approche Agile Kanban. En effet, dès le début du projet, il nous était nécessaire de découper les différentes fonctionnalités du projet en petites issues. Pour cela, nous avons utilisé GitHub Projects pour créer différentes vues qui permettent de créer des petites cartes et de leur affecter des tags, et de créer des issues sur le repository au fur et à mesure du développement de l'application. Voici deux vues différentes du Kanban :





Nous devions aussi préciser la priorité et la difficulté des différentes issues. À partir de là, chacun a alors pu choisir les issues sur lesquelles il voulait travailler.

Pour chaque issue, une PR devait être ouverte, et vérifiée par au moins deux autres personnes. Sur la fin du projet, ce nombre s'est réduit à une personne pour accélérer les cycles.

4 Persistence

4.1 Choix de l'ORM

Plusieurs choix se sont offert à nous quant au choix de l'ORM, dont Exposed et Room. Exposed est un ORM développé par Jetbrains et spécifiquement pour Kotlin et ses coroutines, dont certains d'entre nous avaient déjà l'habitude. Room est un ORM spécialisé pour Android, mais a été pensé en priorité pour Java, et possède une API beaucoup plus archaïque. Nous avons fini par nous rabattre sur Room, par soucis de méconnaissance générale d'Android, et de sa capacité à supporter toutes les features de Exposed.

Ce choix n'est pas regretté, mais l'ergonomie de la librairie n'est pas optimale comparée à d'autres outils connus du marché tels que Exposed ou encore Hibernate. En effet, la gestion des mappings, et en particulier des tables de jointures et clés étrangères est très primitive, et doit être gérée manuellement en tout point, perdant alors en grande partie l'intérêt d'un ORM, et étant alors prône à des erreurs.

4.2 Mapping

L'application possède sept tables différentes, et qui ont des relations entre elles. Ces tables sont décrites par des entités qui sont des data class dont les clés primaires sont des UUIDv4 (assurant leur unicité dans un sens probabiliste).

Un Book est défini par son ID, et un nom qui est une simple String. Il est à noter qu'on doit être capable de récupérer tous les books, ou d'en sélectionner un par son ID

ou son nom, en plus du reste des opérations CRUD. Chaque book est un point central de toutes les autres relations, car toutes les autres entités se réfèrent à un et un seul book. Un book est unique par son nom.

Une Transaction est liée à un book par une clé étrangère sur son ID, et possède elle même un ID, une description, une date, un montant (positif ou négatif), une devise (qui n'est pas utilisée), ainsi qu'une latitude et longitude optionnelles.

D'autre part, une transaction peut posséder une liste d'Attachment. Un attachment est lié à une transaction par une clé étrangère sur son ID, et possède elle même un ID, un nom, et une URI.

Il est possible de créer des Tags pour un book. Un Tag est lié à un book par une clé étrangère sur son ID, et possède lui même un ID, un nom, un type sémantique (`INFO`, `INCOME`, `EXPENSE`, `TRANSFER`), et un budget target optionnel. Un Budget Target contient une valeur, une devise, ainsi qu'une période temporelle (journalière, hebdomadaire, mensuelle, annuelle). Un tag est unique par son nom, son type, et le book auquel il appartient. De cette façon, il est possible, pour un book, de créer deux tags qui ont le même nom, mais des types sémantiques différents.

Puisqu'il est possible d'assigner des tags à une transaction (relation many to many), il est nécessaire d'avoir une table de jointure, qui fait le lien entre les deux tables grâce à des clés étrangères.

D'autre part, il doit être possible pour l'utilisateur de sauvegarder des filtres de recherche pour les recharger immédiatement au besoin. Un filtre est lié à un book par une clé étrangère sur son ID, et possède elle même un ID, un nom, et un champs JSON qui lie chaque critère par son nom à sa valeur.

Enfin, il est aussi possible de filter par tags, et par conséquent, il est nécessaire d'avoir une table de jointure pour lier des tags à des filtres.

4.3 Importation / Exportation

4.3.1 Sérialisation

Avant de pouvoir exporter des carnets de compte dans des fichiers, il est nécessaire de pouvoir les sérialiser, c'est à dire les écrire dans un format texte bijectif (objet => text => object). Pour cela, nous avons utilisé la librairie `kotlinx.serialization`, développée par Jetbrains. Il suffit d'annoter des data class de l'annotation `@Serializable` pour pouvoir la sérialiser en JSON. Les data class en question sont des DTO qui sont un exact miroir des entités en DB, avec la différence que les relations sont gérées avec de la composition classique.

4.3.2 Chiffrement

Il était aussi demandé de pouvoir chiffrer un fichier grâce à un mot de passe. Pour cela, il a suffit d'utiliser un des algorithmes AES/CBC que Java fourni dans son package `javax.crypto`, et de se servir du mot de passe comme clé de salage ainsi que paramètre IV. Ainsi, passer le JSON sérialisé à travers l'algorithme de chiffrement produit le

bytearray illisible attendu.

4.3.3 Upload / Download

Pour l'upload et le download des fichiers d'export book, nous avons créé un microservice Ktor avec deux routes simples :

- `POST /api/store/book` qui attends un multipart form data avec une clé "file" qui a pour valeur les octets du fichier à upload, et renvoie un JWT unique au fichier.
- `GET /api/store/book` qui attends un header Authorization Bearer token qui contient le JWT du fichier que l'on veut récupérer, et qui renvoie les octets du fichier correspondant.

Ktor est une librairie Kotlin développée par Jetbrains et permet la création rapide de clients et serveurs HTTP. Heureusement, certains d'entre nous avaient déjà travaillé sur un projet l'utilisant, nous avons donc pu réutiliser quelques un de nos fichiers pour créer le microservice de Plutus. (Repository de Hedera)

4.4 Workflow d'une exportation

Lorsqu'un utilisateur souhaite exporter un carnet, il doit se rendre sur ce dernier, puis sélectionner "Export book" dans le menu contextuel. Une fenêtre de dialogue apparaît pour demander diverses informations et notamment si l'utilisateur souhaite exporter son carnet sur le service d'hébergement dans le cloud, le nom du fichier créé, ainsi que le mot de passe utilisé pour chiffrer le fichier.

Lorsqu'un fichier et uploadé sur le cloud, le serveur répond une clé signée unique qui permettra d'accéder au fichier pour l'importer depuis l'application. La clé est directement copiée dans le presse-papier de l'appareil, prête à être collée quelque part pour ne pas la perdre. Le carnet et tout son contenu sont exporté, à l'exception des transactions, qui ne se voient exportées que si elles respectent actif au moment de l'exportation. Par exemple si l'utilisateur active un filtrage par étiquette, montant ou autre critère, seuls les transactions respectant ces critères (et donc celles-affichées au moment de l'exportation) seront pris en compte.

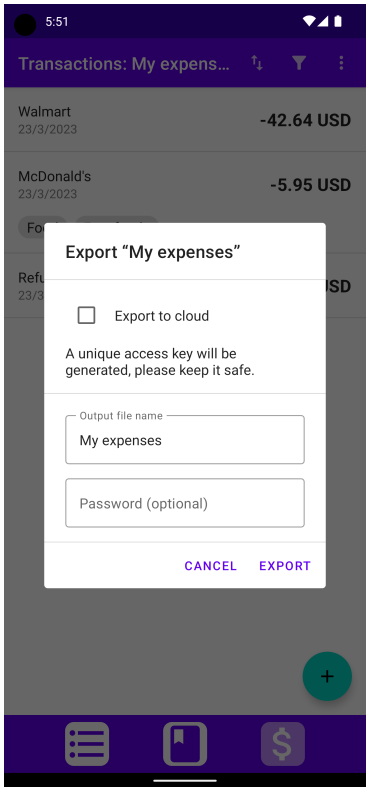


FIGURE 1 – Formulaire d'exportation

Le fichier est ensuite stocké soit dans le stockage de l'appareil, soit dans le cloud via le microservice. Le fichier ainsi créé contient les informations du carnet, ainsi que toutes ses transactions. Les fichiers attachés ne sont cependant pas exportés.

4.5 Workflow d'une importation

Lorsqu'un utilisateur souhaite importer un carnet, il doit dans un premier temps créer un carnet vide dans l'application, afin d'y importer le contenu d'un autre. Pour importer un fichier, il faut choisir "Import book" dans le menu contextuel. Une fenêtre de dialogue apparaît pour choisir si le fichier doit être importé depuis le cloud, et pour donner sa clé le cas échéant ; et le mot de passe pour décrypter le fichier si nécessaire.

N.B : Si un carnet est ré-importé dans lui-même, alors les transactions et autres éléments composant le carnet sont mis à jour. Pour se faire, L'ID du carnet est utilisé afin de savoir si l'importation est un ajout ou une fusion.

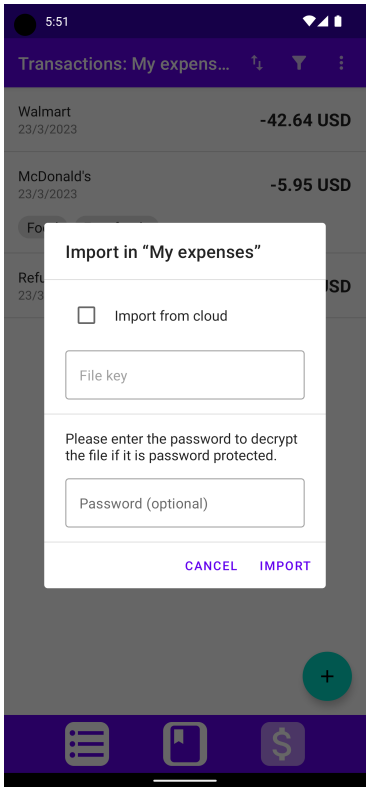


FIGURE 2 – Formulaire d'importation

5 Affichage

5.1 Global state

Un problème récurrent au développement frontend est la gestion de l'état global de l'application. En effet, que nous utilisions des frameworks web tels que Vue ou React, ou bien Jetpack Compose, le problème est le même : comment avoir un état global à l'application et qui est aussi réactif.

La première question qui se pose est "Pourquoi avoir besoin d'un tel état global ?". En effet, il est parfois nécessaire de transmettre à toute l'application des données courantes. Une façon de faire, peu maintenable, et de faire descendre des informations par des props ou des paramètres, de les faire remonter l'arbre de composants grâce à des events ou des callbacks. Bien que cela soit approprié à l'échelle de la communication entre deux components, ce n'est plus le cas, cela signifierait créer des dépendances très fortes entre beaucoup de components qui ne sont pas directement liés, en plus d'avoir beaucoup de paramètres inutiles qui ne serviront qu'à être délégués.

Le global state pattern résout exactement ce problème. Par exemple, savoir quel est le book courant à travers toute l'application. La difficulté étant de savoir comment le rendre réactif.

Pour cela, nous avons crée une paire de hooks (ou composables fonctionnels) qui permettent d'extraire la logique de l'obtention et la gestion de ce global state. En l'occurrence, le global state est une interface contenant de simple propriétés mutables, par

exemple `currentBook`.

Le composable `initGlobalState()` est à lancer une seule fois dans la durée de vie de l'application, et initialise le global state avec un objet anonyme qui instancie l'interface mentionnée précédemment. Chacun des champs est délégué par un remember(Saveable) qui est lui même réactif.

Ainsi, lorsque l'on récupère le global state grâce au composable `globalState()` n'importe où dans l'application, il est possible de muter les champs de l'interface et voir les changements se répercuter partout dans l'application où il y avait des dépendances sur ces champs, que cela soit dans de la vue, ou des clés de LaunchedEffect.

```
private lateinit var globalState: GlobalState

interface GlobalState {
    var currentBook: Book?
    var currentTransaction: Transaction?
    var currentView: View
    var writeExternalStoragePermission: Boolean
}

@Composable
fun initGlobalState(): GlobalState {
    val context :Context = LocalContext.current
    globalState = object : GlobalState {
        override var currentBook: Book? by rememberSaveable { mutableStateOf( value: null) }
        override var currentTransaction: Transaction? by rememberSaveable { mutableStateOf( value: null) }
        override var currentView: View by rememberSaveable { mutableStateOf(View.BOOK_SELECTION) }
        override var writeExternalStoragePermission: Boolean by rememberSaveable {
            val permission :Int = ActivityCompat.checkSelfPermission(
                context,
                Manifest.permission.WRITE_EXTERNAL_STORAGE,
            )
            mutableStateOf( value: permission == PackageManager.PERMISSION_GRANTED) ^rememberSaveable
        }
    }

    return globalState
}

@Composable
fun globalState(): GlobalState = globalState
```

FIGURE 3 – Composables du Global State

5.2 Scaffold de l'application

Au fur et à mesure du développement du frontend de l'application, nous nous sommes assez vite rendu compte de la difficulté d'organiser les différentes parties de l'application en composants bien rangés, et notamment pour la navigation.

Au début, nous utilisions plusieurs composants Scaffold à travers l'application, quand les différentes vues n'étaient pas encore liées, ce qui dupliquait beaucoup de code.

Lorsque l'on a eu accès au global state réactif, nous avons alors eu l'idée d'avoir un système de vues déclarées en tant que variantes d'une enum class. Cette enum possède alors plusieurs champs qui sont des composables et qui correspondent aux différentes parties du scaffold. Ainsi, dans le global state, nous pouvons stocker la valeur d'enum correspondant à la vue courante. Lorsque l'on veut changer la vue, il suffit alors de changer la valeur dans le global state. Dans les faits, c'est un comportement très proche de React Router ou Vue Router.

En voilà un exemple :

```
enum class View(
    val headerComponent: @Composable () -> Unit,
    val contentComponent: @Composable (PaddingValues) -> Unit,
    val fabComponent: @Composable (() -> Unit) = {}
) {

    BOOK_SELECTION(
        headerComponent = {
            TopAppBar(title = { Text( text: "Books") })
        },
        contentComponent = { BookSelectionView() },
        fabComponent = {
            val globalState :GlobalState = globalState()
            FloatingActionButton(onClick = {
                globalState.currentView = BOOK_CREATION
            }) {
                Icon(
                    Icons.Default.Add,
                    contentDescription: "New book"
                )
            }
        }
    ),
```

FIGURE 4 – Déclaration de la vue de sélection de book

FIGURE 5 – Scaffold qui utilise le global state

Ce changement a permit une évolution très rapide du développement de l'application.

5.3 Vues en listes

À plusieurs endroits dans l'application, nous avons besoin d'afficher les éléments sous forme de liste. Les éléments de la liste doivent afficher des informations propres aux entités stockées en base de donnée. Pour ce faire, nous avons défini à chaque fois un composant qui représente un élément unique, ainsi qu'un autre composant qui représente la liste complète, crée par le biais d'une LazyColumn. Les éléments sont toujours composés d'un contenu et du diviseur inférieur.

En plus des informations à afficher, chaque élément de la liste permet d'accéder à la ressource par un simple clic. Ce genre d'interaction est gérée via un callback sur le composant. Ce callback se charge ensuite de modifier la vue courante ainsi que certaines informations dans le `globalState()`.

5.4 Gestion des fenêtre de dialogue

Les fenêtres de dialogue sont toutes façonnées selon un composant créé par nos soins, pour simplifier leur construction, tout en respectant les règles du Material Design. Ce composant comprend un titre, un corps et deux boutons pour annuler ou valider.

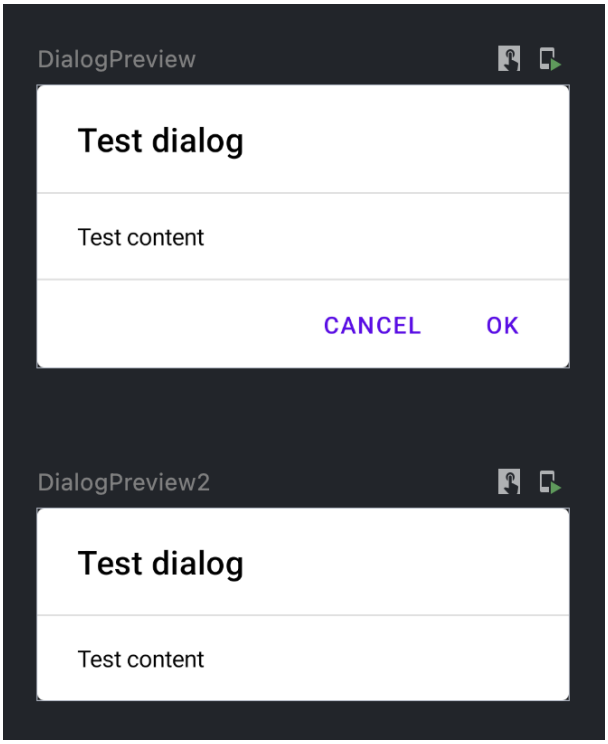


FIGURE 6 – Prévisualisation du composant de Dialog

Certaines fenêtres de dialogue sont notamment ouverte par le biais du `globalState()`, en raison de la séparation entre le composant qui porte la fenêtre et celui qui déclenche son ouverture. Cette limitation est principalement due à l'utilisation du Scaffold, mais est un miroir de la façon dont sont gérée les modales dans la plupart des applications web.

6 Permissions

Afin de proposer certaines fonctionnalités avancées dans notre application, il nous a fallu interagir avec différents composants de l'appareil. En effet, pour l'import/export de Book, ou bien pour la géolocalisation, nous avons dû faire appel à divers éléments du smartphone.

Par souci de confidentialité, Android restreint le champ d'action des applications, et n'offre pas l'accès à toutes les données disponibles. Ainsi, par défaut, il ne nous est pas possible de consulter la position de l'usager, ni d'accéder à ses dossiers sans que ce dernier ne nous y ait autorisé en amont.

6.1 Avertir l'utilisateur

Avant toute chose, il est nécessaire de lister toutes les permissions qui peuvent être demandées par l'application dans le manifest de celle-ci. Cela permet notamment d'indiquer à l'utilisateur dans le marché des applications, quelles sont les autorisations qu'il devra fournir pour tirer pleinement des fonctionnalités de l'application.

De plus, tenir à jour le manifest, permettra en interne de demander l'autorisation d'utiliser cette fonctionnalité.

6.2 Workflow d'une demande

Lorsque l'utilisateur souhaite réaliser une action qui nécessite une autorisation spécifique, cette dernière fait une demande d'autorisation, souvent sous la forme d'un popup natif (c'est-à-dire, géré par le système d'exploitation Android lui-même).

En interne, pour chaque permission, nous avons défini une méthode pour lui demander son approbation, ainsi qu'une variable dans le global state pour stocker le résultat. L'application va dans premier temps devoir vérifier si elle possède déjà l'autorisation nécessaire à la réalisation de sa tâche. Si elle possède déjà la permission, ou si l'utilisateur lui à accordé, elle peut alors poursuivre la réalisation de sa tâche, et débuter les communications avec le composant cible. Dans le cas contraire, elle doit en faire la demande en passant par l'API prévue à cet effet et fournie par Android.

6.3 Expérience utilisateur

Du point de vue utilisateur, lorsque ce dernier effectue une action qui nécessite l'accès à certaines composantes privées de l'appareil, l'application lui ouvre un popup. Ce popup offre plusieurs choix à l'utilisateur, il peut alors décider de lui octroyer la permission pour toujours, le temps de la session d'utilisation, ou bien lui refuser l'autorisation. L'utilisateur pourra à tout moment revenir sur sa décision via les paramètres du téléphone.

6.4 Localisation et écriture sur disque

Les deux principales utilisations de l'API des permissions ont été pour la localisation automatique de l'utilisateur grâce au GPS, lors de la création ou l'édition d'une transaction. En effet, puisque l'on peut localiser une transaction, il était pratique pour l'utilisateur de pouvoir choisir d'affecter automatiquement la localisation de la transaction à sa position actuelle.

D'autre part, pour l'import/export de book sous forme de fichiers, il est nécessaire de les stocker sur l'appareil. Par conséquent, il est nécessaire d'avoir les permissions d'écriture ou de lecture sur un dossier donné.

6.5 Réactivité

Une des difficultés apparues lors du développement de ces features à encore une fois été lié à la façon dont nous devions gérer la réactivité. En effet, naturellement, lorsque l'utilisateur affirme donner l'accord des permissions demandées, le component courant n'est pas rafraîchi, et ne comprends pas que l'action a eue lieu. Ainsi, il était nécessaire pour l'utilisateur de demander une nouvelle fois la réalisation de sa tâche pour que l'application se rende compte qu'elle en possède déjà l'autorisation. Afin de régler ça, nous effectuons la demande de droit, ainsi que l'interaction avec le composant cible au sein d'un LaunchedEffect, avec comme clef l'état de la permission stocké dans le global state. Ainsi, lorsque le programme entre dedans soit il possède déjà la permission et peut continuer le traitement, soit il soumet une demande. Suite à cela, soit il donne son accord, l'état d'approbation change, et le LaunchedEffect se relance et poursuit le traitement, soit il refuse, et le LaunchedEffect s'arrête.

7 Notifications

Dans le cadre de notre projet, nous avons veillé à gérer de manière optimale les notifications liées au budget associé à chaque étiquette. En effet, grâce à notre application, l'utilisateur peut définir un budget spécifique pour chaque étiquette, ce qui lui permet de mieux contrôler ses dépenses et de ne pas se retrouver en situation de dépassement budgétaire.

Ainsi, si l'utilisateur dépasse le budget alloué à une étiquette donnée, notre application envoie automatiquement une notification pour l'en informer et lui permettre de prendre les mesures nécessaires.

En outre, nous avons ajouté une fonctionnalité intéressante à notre application, en l'occurrence l'étiquette @todo qui permet à l'utilisateur de spécifier une transaction qui n'a pas encore été réalisée, mais qui doit l'être à une date ultérieure. Dans ce cas de figure, notre application est en mesure d'envoyer des notifications à l'utilisateur pour le rappeler de la réalisation de cette transaction, en se basant sur la date d'échéance qui a été spécifiée dans l'étiquette @todo.

En somme, nous avons donc pris en compte toutes les situations possibles qui pourraient engendrer des dépassements de budget ou des oublis de transactions, et avons mis en place des mécanismes de notifications efficaces pour aider l'utilisateur à mieux gérer ses finances et à ne rien oublier.

Malheureusement, nous n'avons pas été en mesure d'ajouter les notifications liées à la géolocalisation dans notre application, en raison d'une limitation technique. En effet, nous n'avons pas encore implémenté toutes les fonctionnalités relatives à la géolocalisation lors du déplacement de l'utilisateur, ce qui a rendu impossible l'ajout de cette fonctionnalité de notification.

Un problème notable que nous avons constaté est que l'application ne gère pas les doublons de notifications. En effet, lorsque deux notifications identiques sont reçues en même temps, l'application n'envoie que la dernière notification. Cela peut poser problème pour les utilisateurs qui ont besoin de recevoir toutes les notifications pour s'assurer de ne pas manquer d'informations importantes.

8 Fonctionnalités manquantes

Les devises différentes ne sont pas fonctionnelles. Une partie du backend à été réalisé, mais les différentes conversions qui auraient du être récupérées n'ont pas été faites. Par conséquent, on ne peut choisir que l'USD comme devise.

Les notifications liées à la géolocalisation n'ont également pas été implémentées. Lorsque l'utilisateur se rapproche du lieu d'une de ses transaction, une notification devait lui être envoyée, mais cela n'a pas été fait.

Le Markdown n'est pas géré dans les descriptions de transactions.

Lorsque l'on exporte un book, tout est bien exporté, excepté les attachment liés aux

transactions.

On peut voir la localisation d'une transaction individuelle sur une carte, mais on ne peut pas voir l'ensemble des transactions sur la même carte.

Il n'y a pas de suppression de masse des transactions.

9 Conclusion

Pour conclure, bien que nous soyons globalement satisfaits du travail accompli par l'ensemble du groupe, ce projet nous a tous paru bien trop conséquent.

En effet, l'intitulé de la matière étant "Android **frontend**", nous avions un nombre très conséquent de tâches ayant très peu de lien avec le frontend. Le nombre et la variété des tâches n'entrant pas dans cette catégorie est à la limite de l'absurde, à tel point que nous avons réalisé des tâches que nous n'avons même pas eu à faire dans notre projet de **backend**. De plus, variété mise à part, la quantité de travail impliquée par ce projet est bien trop disproportionnée, sans prendre en compte la contrainte de temps.