SZCZECH Mateusz

SDAF Zineb

THEAS-LABAN Alexis

ZAIANI Yassine

Rapport de développement

Pokédex

Programmation mobile Android

Introduction

Une application mobile Pokédex, contient les détails d'un Pokémon, ses évolutions, ses statistiques, ses talents, etc. Permets d'assembler une équipe et d'analyser les forces et faiblesses de celle-ci. Regrouper des Pokémon en les ajoutant à une liste de favoris. Alterner entre plusieurs profils utilisateurs.

Architecture générale

Organisation des fichiers

Les différents packages du projets:

- **Components** - contient tous les éléments graphiques de type composable.
- **Service** - classes services comme l'estimateur de forces et faiblesses, le lecteur de musique ou le BroadcastReceiver.
- **Data** - contient les classes relatives au données de l'application, le package **user** pour les informations de l'utilisateur contenues en base de données et **pokedex** pour les informations extrait de l'API.
- **Theme** - package de base.
- **Utils** - continent des classes utilitaires.

Import des données csv

Une première étape du projet consiste en la récupération des données concernant les pokémons. Ces données pourront contenir des informations sur les caractéristiques d'un pokémon, ses évolutions, sa nature, etc.

Les classes implémentant l'interface Parser permettent de parcourir les fichiers csv et d'en extraire des informations exploitables pour les autres composants du projets.

Une série de classes de stockage récupère ces listes d'informations et servent d'interface avec les autres services du projet.

Ces classes de stockages sont les suivantes :

- **Move Repository** - pour accéder à l'ensemble des capacités apprises par les Pokémon.
- **Ability Repository** - pour accéder à l'ensemble des talents des Pokémon.
- **Location Repository** - pour accéder aux lieux de rencontre des Pokémon.
- **Evolution Chain Repository** - pour accéder aux différentes évolutions d'une lignée de Pokémon.
- **Pokemon Repository** - pour accéder à l'ensemble des Pokémon et leurs caractéristiques.

Note: certaines classes de stockage ne sont pas utilisées (Move et Location).

Ces classes sont regroupées dans la classe **Pokedex Storage Service**, un objet singleton qui doit être initialisé au démarrage de l'application.

Une première approche pour le stockage des Pokémon était d'utiliser une seule classe **Pokémon Repository** qui assurait un stockage semblable à du document store; les pokémon contenait une référence vers les capacités qu'ils utilisaient, les talents qu'ils possédaient, etc.

Cette implémentation à dû être repensée car les objets Pokémon n'étaient pas Sérialisable et l'application ne pouvait pas tourner en fond de tâche.

L'implémentation actuelle référence des entités distantes par leurs identifiants; un objet Pokémon contient un champ **evolutionChainId** pour référencer l'entité **EvolutionChain** qui contient les évolutions de sa lignée.

Outil pour évaluer les forces et faiblesses d'une équipe

La classe **Team Fact Generator** produit des informations relatives aux équipes en se basant sur deux points principales :

- Les stats de bases de l'équipe
- Les faiblesses / résistances à un certains type

Chaque information d'analyse est catégorisée selon son orientation; négative, positive ou neutre.

Evaluation par statistiques

Une inspection des statistiques de base de l'équipe en se basant sur le total moyen des statistiques.

Deux type d'informations peuvent être générée:

- Une information sur la variance - les statistiques de l'équipe sont déséquilibrées ou au contraire bien réparties.
- Une information sur le total des stats - les statistiques de l'équipe sont trop faibles ou trop élevées.

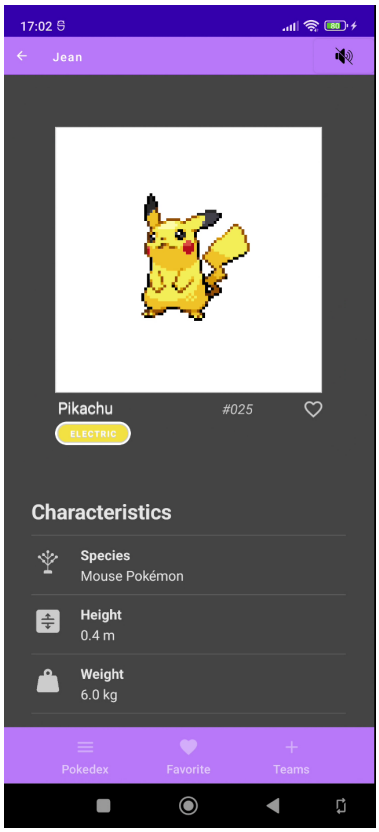
Evaluation par types

Une inspection des faiblesses et résistances de chaque membre de l'équipe, si 3 membres ou plus sont vulnérables ou résistant à un type, une information est produite.

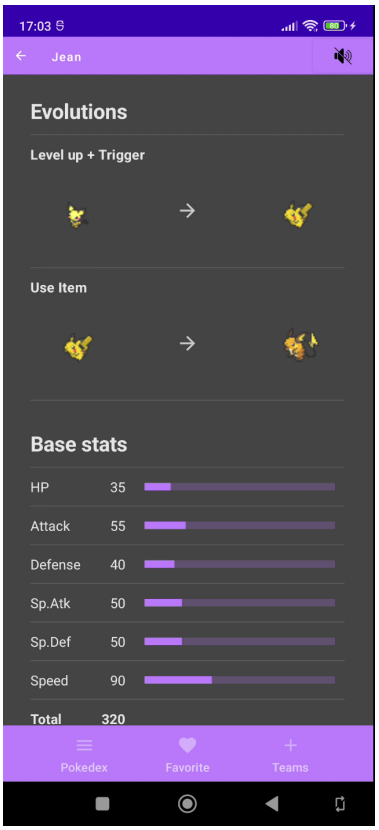
Le double type permet parfois de retirer certaines faiblesses du Pokémon et sont prises en compte dans l'analyse.

Par exemple: un pokémon de type **FIRE WATER** sera neutre face aux attaque de type **GRASS** (résistance du type **FIRE** et faiblesse du type **WATER**).

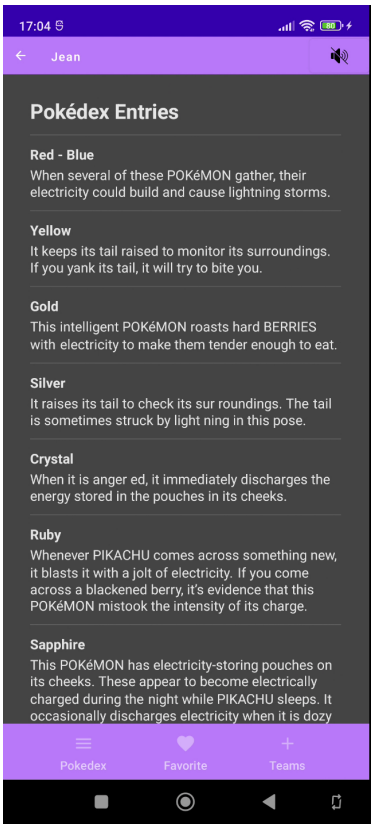
Note: l'immunité à certains types n'a pas été implémentée.



Carte du pokemon Pikachu



Informations utiles du pokemon Pikachu



Descriptions du pokemon Pikachu

Connexion à la base SQLite en utilisant la bibliothèque de persistance room

Nous avons utilisé la bibliothèque de persistance room car elle était recommandée sur la documentation officielle d'android.

L'application stocke dans la base de données les données suivantes :

- Les profils des utilisateurs
- Les pokémon favoris sauvegardées pour chaque profil
- Les équipes de pokémon sauvegardées pour chaque profil

Les données des pokémon ne sont pas stockées dans la base de données car l'application les importe au démarrage grâce aux fichiers de CSV. Chaque pokémon possède son propre identifiant (Long) donc par exemple pour rajouter un pokémon dans les favoris il suffit de stocker son identifiant. Si jamais on veut récupérer un pokémon favori, on récupérera seulement son identifiant pour ensuite récupérer l'objet Pokémon de la hashmap en utilisant son identifiant comme clef.

Plusieurs classes ont été développées pour stocker les données.

- **Profile** qui stocke son id et le nom du profil
- **Favorite** qui stocke son id, l'id du pokémon favori et l'id du profil associé
- **Team** qui stocke son id, le nom de l'équipe et l'id du profil associé
- **TeamMember** qui stocke son id, l'id du pokémon membre de l'équipe et l'id de l'équipe associée

Afin de permettre à Room de récupérer les favoris ou les équipes associées a un profil les classes suivantes ont été développées :

- **ProfileWithFavorites** qui récupère le profil avec la liste des pokémon favori
- **ProfileWithTeam** qui récupère le profil avec la liste des équipes
- **TeamsWithMembers** qui récupère l'équipe avec la liste des pokémon membres de l'équipe

Pour permettre aux développeurs de communiquer avec la base de données, des daos ont été implémentées :

- **FavoriteDao** qui permet d'ajouter ou de supprimer un favori
- **ProfileDao** qui permet d'ajouter/modifier/supprimer/récupérer un profil et de récupérer les équipes et favoris en utilisant les classes ci-dessus
- **TeamDao** qui permet d'ajouter/modifier/supprimer/récupérer une équipe et de récupérer ses membres en utilisant les classes ci-dessus
- **TeamMemberDao** qui permet d'ajouter/supprimer/récupérer un membre d'une équipe

La class **PokedexAppDatabase** permet de récupérer une instance de la connexion a la base de données (singleton) pour ensuite permettre l'utilisation des dao.

Ecran des profils

Au lancement de l'application, l'écran des profils se présente à l'utilisateur.

L'utilisateur est invité à créer son propre profil. Un dialogue apparaît afin que l'utilisateur puisse entrer le nom du profil. Une image aléatoire d'un pokémon est associée au profil afin de permettre une distinction plus facile.

Ensuite une LazyList permet d'afficher tous les profils avec les boutons pour modifier/supprimer un profil.

Une lambda onClick a été implémentée pour chaque profil dans la liste pour permettre à l'utilisateur d'accéder à l'onglet pokédex quand il clique dessus.

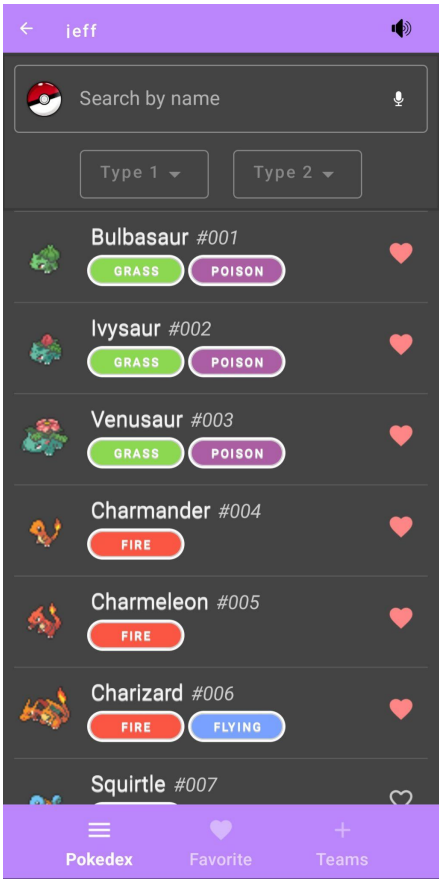
Navigation entre onglets

La barre d’en haut qui permet d’indiquer le profil courant et revenir à l'écran de profils ainsi que la barre d’en bas indiquant les onglets que l’utilisateur peut cliquer ont été réalisées avec des rows et columns. Scaffold gère l’emplacement des barres. AnimatedVisibility a été utilisé pour réaliser une animation d’apparition, disparition des barres (les barres disparaissent à l'écran des profils).

Pour gérer la navigation entre les écrans, un NavHost a été mis en place. Les routes de l’application sont déclarées dans la classe **Route** ce qui permet de facilement les retrouver, d’en rajouter et d'enlever.

NavHost possède plusieurs composables avec leurs routes qui permettent de naviguer et d’afficher un écran en question. Pour communiquer entre les écrans, il est possible de passer des arguments à un écran en les ajoutant directement dans la route. Des navAgrument ont été mis en place pour récupérer les arguments des routes et leur donner un type.

Onglet Pokedex



Onglet Pokédex avec les barres du haut et du bas

Pour arriver à cet écran, c'est le composable de l'onglet Pokédex dans NavHost qui est appelé.

Pour avoir la liste des pokémon sous forme de grille, une `LazyVerticalGrid` est appelée.

Pour afficher de cette manière une cellule de la grille (cf photo Onglet pokedex), on utilise la méthode `PokemonListDisplay` qui affiche l'icône, le nom, l'id, le(s) type(s) du pokémon ainsi que l'icône favori qui est vrai ou non en fonction des choix de l'utilisateur.

Lors d'un redémarrage de l'application, les favoris sélectionnés par l'utilisateur seront toujours affichés (❤️) grâce à la récupération de la liste des favoris en fonction des éléments sauvegardés dans la BDD de chaque profil respectifs.

```
PokedexAppDatabase.getConnection(context).profileDao().getProfileWithFavorites(profilId).favorites
```

Lorsque l'utilisateur effectue un clic sur l'icône favori, on regarde dans la BDD si le pokémon que l'on regarde est en favori ou non grâce à l'id de celui-ci.

Si ce pokémon est renseigné dans les favoris de la BDD du profil concerné, alors cela mène à la suppression de celui-ci dans la BDD des favoris.

```
PokedexAppDatabase.getConnection(context).favoriteDao().deleteFavorite(favorite)
```

Si ce pokémon n'est pas trouvé dans les favoris de la BDD du profil concerné, celui-ci sera ajouté par son id et sa liste des favoris sera mise à jour.

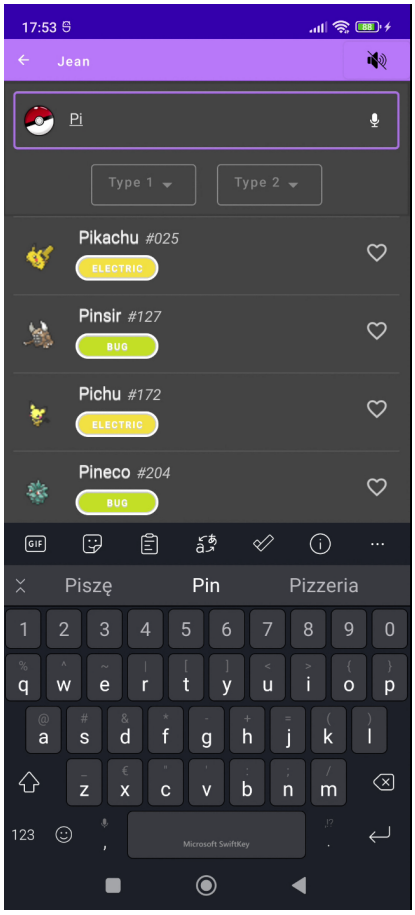
```
PokedexAppDatabase.getConnection(context).favoriteDao().addFavorite(Favorite(pokemonId, profile.getId()))
```

```
PokedexAppDatabase.getConnection(context).profileDao().getProfileWithFavorites(profileId).favorites
```

Barre de recherche

Recherche par chaîne de caractère

En cliquant sur la barre de recherche, l'utilisateur voit le clavier s'ouvrir et lui permettre d'inclure du texte. Pour fermer le clavier virtuel, il suffit d'utiliser les fonctionnalités de l'Android.



Recherche par nom d'un pokémon commençant par "Pi"

Cela est possible grâce à `OutlinedTextField`, ce composable permet à l'utilisateur d'inclure du texte qui sera récupéré par la suite.

Dès que l'utilisateur rentre un caractère, celui-ci est récupéré dans une lambda qui va permettre par la suite d'avoir une liste de pokémon filtrée en fonction de la chaîne de caractère donné par l'utilisateur. Même si l'utilisateur n'indique pas de majuscule au nom de pokémon qu'il essaye de trouvé, la première lettre de son texte sera mise en majuscule. Ce qui permet un meilleur filtrage.

```
resultList = resultList.filter { it.name.contains( if (search.isNotEmpty())
search.replaceFirst( search.first(), search.first().uppercaseChar()) else
search)}.toMutableList()
```

Recherche vocale

En touchant l'icône du microphone, l'application va demander les droits d'accès à celui-ci. Si l'utilisateur accepte, il pourra alors prononcer à haute voix le nom d'un pokémon pour faire une recherche vocale.

Pour demander l’autorisation à l'utilisateur on utilise le dialog:

```
rememberLauncherForActivityResult(ActivityResultContracts.RequestPermission())
```

Et on l’affiche en lui donnant le type de permission voulu avec:

```
.launch(android.Manifest.permission.RECORD_AUDIO)
```

Pour tester si la permission est accordé ou non nous utilisons la condition suivante:

```
ContextCompat.checkSelfPermission(applicationContext,
android.Manifest.permission.RECORD_AUDIO) == PackageManager.PERMISSION_GRANTED
```

Une fois la permission accordée, un dialogue apparaît pour donner signe à l'utilisateur que l'application est prête à l'écouter.

Pour réaliser cette feature nous avons utilisé la classe SpeechRecognizer:

```
SpeechRecognizer.createSpeechRecognizer(applicationContext)
```

Pour ensuite utiliser la méthode:

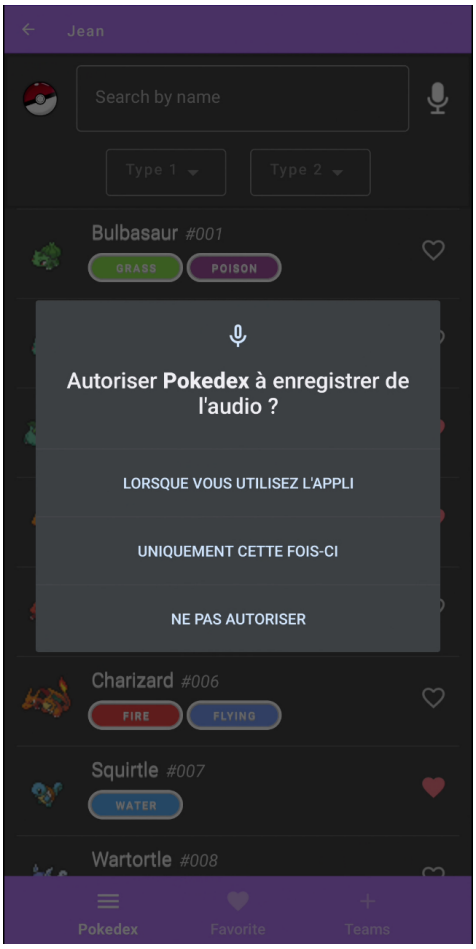
```
setRecognitionListener()
```

Qui prend en paramètre l’implémentation anonyme de l’interface **RecognitionListener**. Une chaîne de caractère va alors être produite après que l’utilisateur a fini de parler dans la méthode:

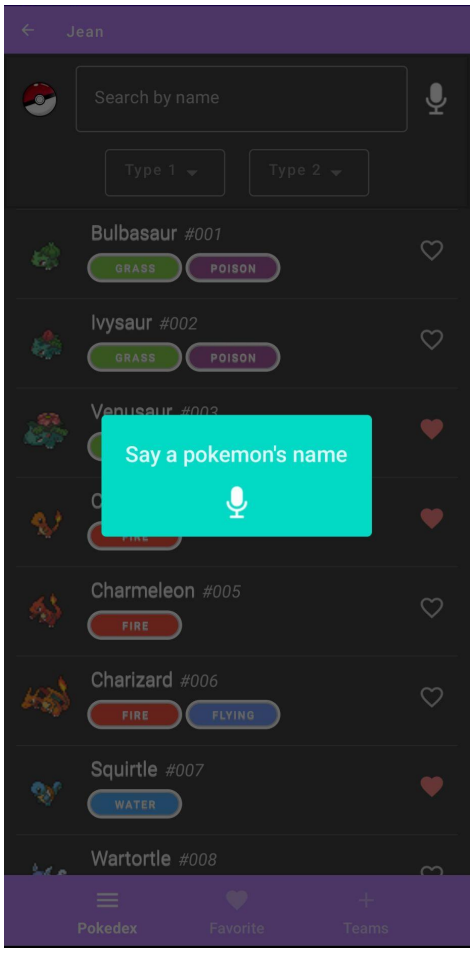
```
onResults()
```

Il suffit donc d'insérer ce String dans le champ filtre pour filtrer les Pokémon.

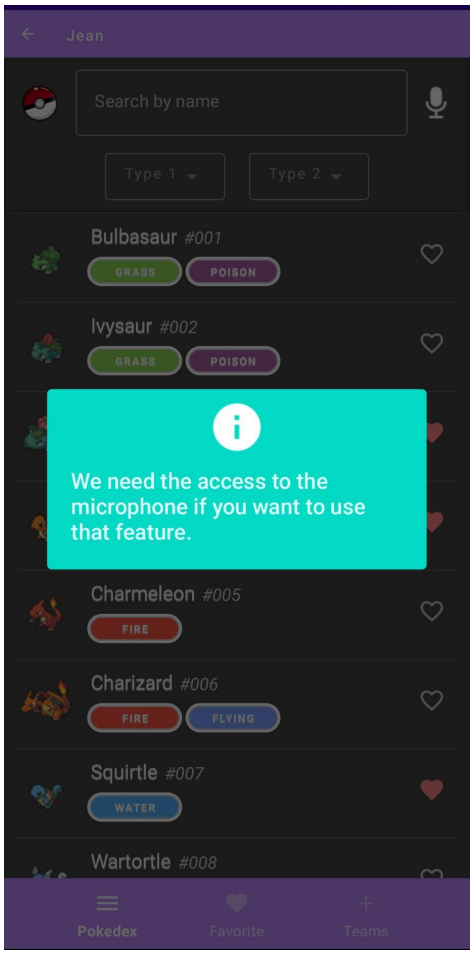
Si l'utilisateur refuse, l'application lui expliquera qu'il n'est pas possible d'utiliser cette feature sans autorisation d'accès au microphone.



Demande d'autorisation au microphone



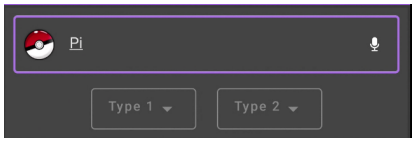
Message indiquant à l'utilisateur que l'application est prête à l'écouter



Message d'information indiquant qu'il est impossible d'utiliser la feature

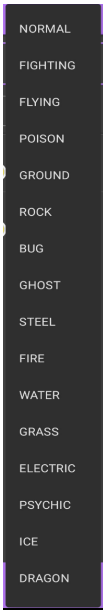
Filtre Type

En dessous de la barre de recherche, on peut voir deux box qui portent le nom de Type 1 et Type 2.



Barre de recherche et box des filtres de type

Ces box permettent d'appliquer des filtres cette fois-ci sur les types des pokémon.



Menu déroulable des différents types existant

Ce sont deux menus déroulant scrollable (`DropdownMenu`) qui affiche les différents types existants et chacun d'eux est cliquable grâce à `DropdownMenuItem`.

Lorsque l'utilisateur clique sur un des types proposés dans les box correspondantes, le type choisi est récupéré dans une lambda.

```
DropDownType(name = "Type 1", typeNum = { type1 = it })

DropDownType(name = "Type 2", typeNum = { type2 = it })
```

Ensuite, on parcourt la liste des pokémon pour trouver ceux qui respectent la condition du type imposé en regardant dans les deux types de chaque pokémon, pour les ajouter dans une autre liste.

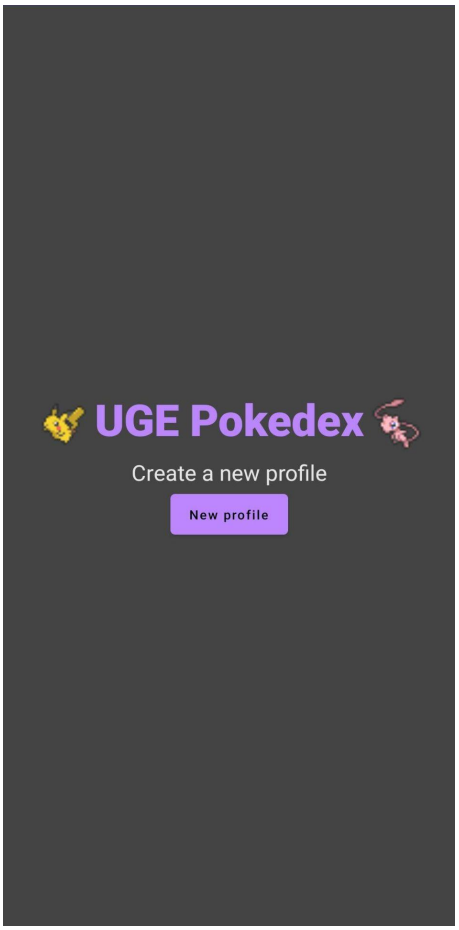
```
for (pokemon in pokemonList) {

   if ((pokemon.type.first.toString().contains(type1) ||

        pokemon.type.second.toString().contains(type1)) &&

      (pokemon.type.first.toString().contains(type2) ||

        pokemon.type.second.toString().contains(type2))) {

                resultList.add(pokemon)

            }

        }
```

Après chaque changement sur les types désirés par l'utilisateur et les caractères donnés dans la barre de recherche, un `LaunchedEffect(type1, type2, search)` est effectué pour filtrer de nouveaux en fonction des changements dans la méthode `FilterBar` qui prend en compte les deux filtres.

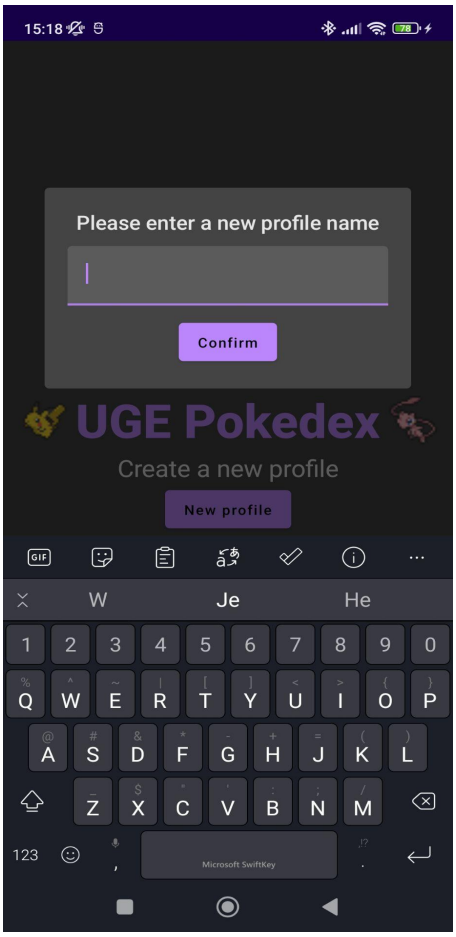
La liste qui contient les filtres (types et recherche) est envoyée dans une lambda et sera appelée dans la méthode `Pokemon List Display` pour afficher le pokédex filtré.

```
FilterBar(pokemonList = pokemonMap.values.toList(), applicationContext = applicationContext) {

      filteredPokemon = it.toMutableList()}

PokedexDisplay(pokemonList = filteredPokemon...)
```

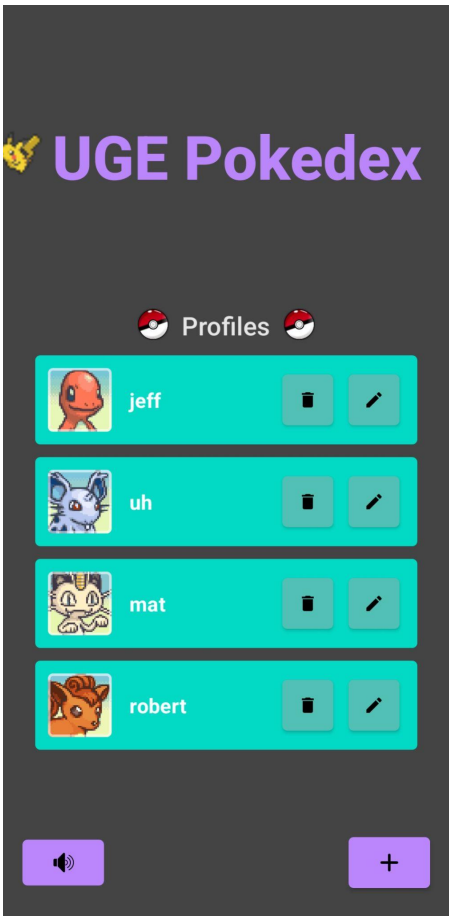
Carte pokemon



Écran de profil au démarrage de l'application



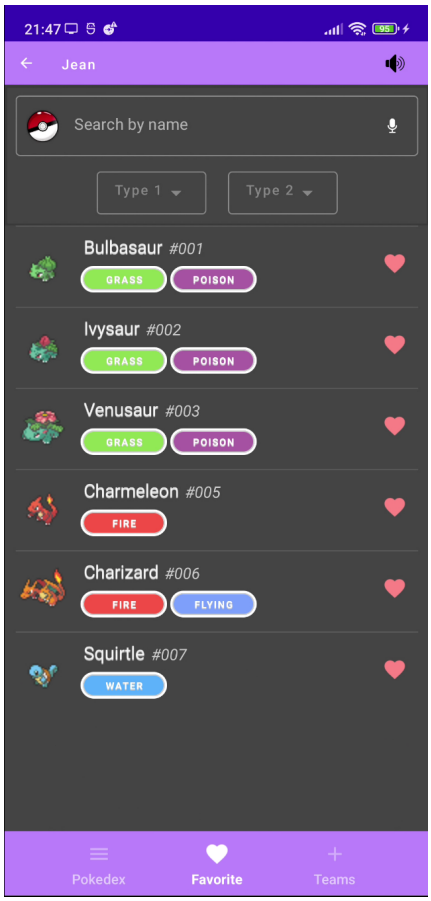
Dialog qui permet de saisir un nom de profil



Ecran de profil après avoir créer plusieurs profils

Onglet Favorite

Dans cet onglet on retrouve à peu près les mêmes fonctionnalités que dans l'onglet Pokédex, à quelques différences près.



Onglet favoris

Liste des favoris

Contrairement à la liste des pokémon dans l'onglet Pokédex, il n'y a que les pokémon mis en favoris par l'utilisateur. La méthode `PokemonlistDisplay` affichera donc les éléments de cette liste qui est par la suite envoyé dans une lambda de la méthode `FilterBar` :

```
pokemonList = favoritesPokemon.distinct()
```

Il peut avoir un œil sur la totalité de ses favoris, appliqué des filtres, mais aussi désélectionné des favoris.

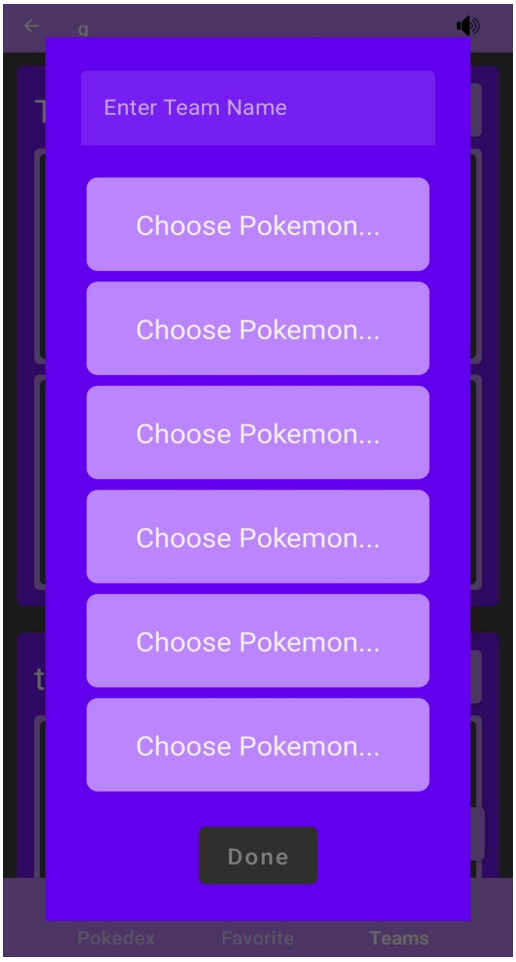
Si l'utilisateur choisit de ne plus garder en favoris un pokémon, le même principe s'effectue `comme` lors de la désélection de pokémon dans l'onglet Pokédex.

Après un retrait de favori, celui-ci reste affiché tant que l'utilisateur ne change pas d'onglet pour éviter les suppressions non volontaires.

Écran des équipes

Dans l'onglet équipe l'utilisateur peut cliquer sur le bouton "+" cela va ouvrir un dialog qui va lui permettre de créer son équipe, la sélection des Pokémon se fait à partir du Pokédex, l'équipe créé est associée au profil courant et est ensuite enregistrée dans la base de données (voir BDD).

Onglet Équipe vide



Dialog de création d'une équipe

Édition d'une équipe

Lors de l'édition d'une équipe, on se contente de supprimer tous les membres de l'équipe que l'on souhaite modifier et d'ajouter les nouveaux Pokémon, on ne crée pas de nouvelle équipe.

Les équipes sont affichées à l'aide d'une LazyVerticalGrid chaque équipe est indexée avec `itemsIndexed`, index qui va servir si un nom d'équipe n'est pas choisi, l'équipe portera alors le nom par défaut : `"Team $index"` .

Lorsque que l'on clique un pokémon d'une équipe, sa carte s'affiche avec ses informations de la même manière que le Pokédex.

Chaque `Composable` servant à afficher une équipe est cliquable, si l'on clique sur le nom d'une équipe, cela appelle un autre `Composable` responsable de l'affichage des pokémons de l'équipe ainsi que les forces et faiblesses de celle-ci.

Service Musique

Pour améliorer l'expérience utilisateur, nous avons décidé d'intégrer un service de musique.

Il y a deux fichiers qui sont liés à ce service de musique.

Le premier fichier, `PokemonMusicService`, est une classe qui étend la classe `Service` d'Android. Le deuxième fichier, `MusicButton`, contient une fonction `@Composable` qui affiche un bouton qui permet de mettre en pause ou de reprendre la musique.

PokemonMusicService

La classe `PokemonMusicService` est un service Android qui permet de jouer la musique en fond de l'application. Il est possible de contrôler la lecture de la musique à l'aide de différentes méthodes exposées par cette classe.

Les variables :

- La variable `mediaPlayer` est une instance de la classe `MediaPlayer` qui est utilisée pour jouer la musique.
- La variable `binder` est une instance de la classe `LocalBinder` qui est utilisée pour connecter le service à l'activité.
- La variable `state` est un booléen qui indique si la musique est en cours de lecture ou non.

Les méthodes :

`onBind() :` Cette méthode est appelée lorsque le service est connecté à l'activité. Elle renvoie l'objet `binder` pour permettre la communication avec l'activité.

`onCreate()` : Cette méthode est appelée lorsque le service est créé. Elle initialise le `MediaPlayer` avec la musique et définit les auditeurs pour la préparation et la fin de la lecture de la musique.

`onPrepared()` : Cette méthode est appelée quand le `MediaPlayer` est prêt à jouer la musique. Elle lance la lecture de la musique et définit l'état du service à `true`.

`onStartCommand()` : Cette méthode est appelée lorsque le service est démarré. Elle définit l'état du service à `true` et renvoie la constante `START_STICKY` qui permet de redémarrer le service en cas d'arrêt inattendu.

`onDestroy()` : Cette méthode est appelée lorsque le service est détruit. Elle arrête la lecture de la musique, libère les ressources utilisées par le `MediaPlayer` et définit l'état du service à `false`.

`getState()` : Cette méthode renvoie l'état actuel du service.

`setState()` : Cette méthode inverse l'état actuel du service.

`getMediaPlayer()` : Cette méthode renvoie l'objet `MediaPlayer` utilisé par le service.

`LocalBinder` : Cette classe interne est utilisée pour lier le service à l'activité.

Les listener :

`OnPreparedListener` : Cet auditeur est appelé lorsque le `MediaPlayer` est prêt à jouer la musique.

`OnCompletionListener` : Cet auditeur est appelé lorsque la lecture de la musique est terminée. Il redémarre la lecture de la musique et définit l'état du service à `true`.

MusicButton

La fonction `@Composable MusicButton` est utilisée pour afficher un bouton qui permet de mettre en pause ou de reprendre la musique. Cette fonction prend deux paramètres : `audioState` et `onClick`.

En résumé, ces deux fichiers sont utilisés ensemble pour permettre à l'utilisateur de contrôler la lecture de la musique du jeu Pokémon tout en utilisant l'application.

Le fichier `PokemonMusicService` gère la lecture de la musique dans un service en arrière-plan, tandis que `MusicButton` fournit un bouton pour permettre à l'utilisateur de mettre en pause ou de reprendre la lecture de la musique.

BroadcastReceiver

Dans AndroidManifest.xml, la balise **<receiver>** a été ajoutée pour permettre d'envoyer des notifications en fonction de chaque contexte où l'utilisateur peut agir.

Voici les différentes actions reconnues par l'application :

- <action android:name="profileCreated"></action> : l'application peut recevoir et traiter une action lorsqu'un profil utilisateur est créé
- <action android:name="profileEdited"></action> : l'application peut recevoir et traiter une action lorsqu'un profil utilisateur est modifié
- <action android:name="profileDeleted"></action> : l'application peut recevoir et traiter une action lorsqu'un profil utilisateur est supprimé
- <action android:name="favoriteAdded"></action> : l'application peut recevoir et traiter une action lorsqu'un favori est ajouté
- <action android:name="favoriteDeleted"></action> : l'application peut recevoir et traiter une action lorsqu'un favori est supprimé
- <action android:name="teamCreated"></action> : l'application peut recevoir et traiter une action lorsqu'une équipe est créée
- <action android:name="teamEdited"></action> : l'application peut recevoir et traiter une action lorsqu'une équipe est modifiée
- <action android:name="teamDeleted"></action> : l'application peut recevoir et traiter une action lorsqu'une équipe est supprimée.
- <action android:name="teamShared"></action> : l'application peut recevoir et traiter une action lorsqu'une équipe est partagée

Ensuite, une classe PokedexReceiver a été confectionnée : elle sert à recevoir les différentes actions envoyées par l'application.

Dans cette classe qui reçoit des broadcast, on retrouve la méthode :

```
override fun onReceive(context: Context?, intent: Intent?)
```

Elle permet de récupérer, à partir de l'action envoyée dans le context, la clé et le message attribuée grâce à l'utilisation de `.getStringExtra("message")`, puis permet l'envoi d'un toast avec le message correspondant pour notifier l'utilisateur de l'action qui vient de s'effectuer :

```
Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
```

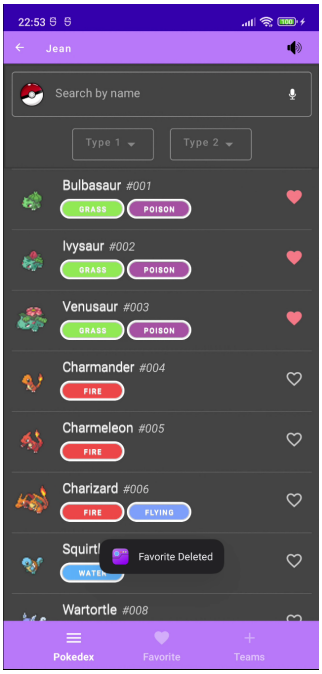
Mais avant de recevoir des actions, un objet `PokedexReceiver` qui reçoit des broadcasts envoyés et un objet `IntentFilter` qui définit les types d'actions que PokedexReceiver peut recevoir ont été créés dans l'activité principale.

L'objet IntentFilter définit toutes les actions détaillées plus haut.

L'appel à la méthode `registerReceiver(receiver, intentFilter);` permet d'enregistrer l'objet PokedexReceiver et permet à celui-ci de traiter les actions définies dans l'objet IntentFilter.

La méthode `fun newIntent(context: Context, action: String, message: String);` définie comme companion object permet de l'appeler un peu partout dans les différentes classes où c'est nécessaire. Elle prend en argument le contexte courant, l'action que l'on souhaite envoyer et le message désiré qui sera affiché.

L'action est désignée à un objet Intent, auquel on ajoute un message de type String à la clé message grâce à `.putExtra("message", message).` Puis sur le contexte, on envoie le broadcast de l'objet Intent créer.



Pour avoir cette notification, voici le code utilisé pour la méthode newIntent :

```
PokedexReceiver.newIntent(context,"favoriteDeleted", "Favorite Deleted")
```

Notification après avoir désélectionné un pokemon des favoris

Répartition des tâches

Mateusz

- Connexion à la base SQLite en utilisant la bibliothèque de persistance room
- Écran des profils
- Navigation entre onglets
- Recherche vocale

Zineb

- Onglet Pokédex
- Onglet Favori
- Barre de recherche avec filtre des Pokémon par type
- broadcastReceiver

Alexis

- Import des données csv
- Composant pour afficher un Pokémon dans une liste
- Affichage des infos diverses d'un Pokémon
- Outil pour évaluer les forces et faiblesses d'une équipe

Yassine

- Onglet équipe
- Service pour écouter de la musique
- Affichage des forces et des faiblesses d'une équipe

Difficultés rencontrées

Import des icônes, sprites

Les icônes récupérées à partir de sites externes présentait une nomenclature qui ne nous permettait pas de les importer directement sur Android Studio.

Exemple de fichier : 1.png, 757-mega.png, 493-bug.png.

Où bug référence le type insecte et non une catégorie de drawable. Pour répondre à cette problématique, il a fallu effectuer une série d'opérations sur les fichiers à l'aide de script :

- Remplacement des tirets par des underscores.
- Ajout d'un ou plusieurs caractères devant le nom du fichier.

Démarrage lent

Le chargement des données de l'api ralentissait considérablement le démarrage de l'application. L'une des raisons de ce ralentissement était la taille de certains fichiers qui pouvaient atteindre plusieurs centaines de milliers de lignes.

Une solution retenue a été de parcourir les données et de supprimer les informations redondantes ou non utilisées (description par langage ou version de jeu, etc) avec des scripts python. Par ailleurs, certaines données n'ont simplement pas été utilisées ; les capacités d'un Pokémon et ses lieux de rencontres.

Mettre l'application en arrière-plan

Une précédente implémentation du stockage des données de l'api nous empêchait de mettre l'application en arrière-plan. En effet, les objets Pokémon n'étaient pas sérialisables et ne pouvaient donc pas être conservés.

Pour résoudre ce problème, une refonte du stockage de ces informations a été nécessaire. Un objet Pokémon référence désormais une autre entité par son id plutôt que par référence d'objet. Par ailleurs, le composant **NavigationGraph** prenait en paramètre la liste des Pokémon, cette implémentation n'étant pas recommandée, elle a été modifiée pour que les composants accèdent directement aux données via le singleton **PokédexStorageService**.

Rafraîchissement des cœurs dans la liste des pokémons

Quand l'utilisateur essayait de rajouter un pokémon en favori, le cœur ne se remplissait pas, il fallait défiler la liste jusqu'à ce que le pokémon en question ne soit plus visible et défiler en arrière pour rafraîchir le pokémon et ainsi voir le cœur rempli. Le bug était difficile à trouver, la LazyList ne possédait pas de clef unique pour chaque pokémon ce qui empêchait le

rafraîchissement des sous composants. La définition d'une clef unique - l'id du pokémon a réglé le problème.

Maintien de l'état du lecteur de musique

Lorsque que l'application n'est pas au premier-plan, la musique se coupe et recommence depuis le début lorsque l'on revient sur l'appli. Je n'ai pas réussi à pallier ce problème, mais j'ai fait en sorte de synchroniser l'état de l'icône audio à l'état du lecteur.