

Application mobile de formation



Crée-moi une application mobile sous Android Studio avec Kotlin, Jetpack Compose et Firebase. Cette application doit pouvoir proposer des cours/formation en ligne se basant sur l'application schoolmouv.

Voici le sujet du TP :

Projet UE HMIN205 : Programmation Mobile.

MeFormer (Ou MySchool, etc.)

Cahier des charges et fonctionnalités

L'objectif de ce projet est de créer une application mobile sous Android pour proposer des cours/formation en ligne. L'application schoolmouv constitue un exemple intéressant <https://www.schoolmouv.fr/>, <https://youtube.schoolmouv.fr/> à suivre pour développer cette application.

Ainsi, cette application doit permettre :

- De présenter les fonctionnalités de cette application.
- De s'inscrire suivant plusieurs profils.
- De bénéficier des services fonctionnels de cette application en fonction du profil de connexion et en fonction du mode de connexion : online, offline.

Présentation de l'application. Une fois l'application installée et sans inscription, elle permet à son utilisateur de bénéficier de :

- Présentation claire et ergonomique de l'application sous différents formats : vidéo, texte, etc.
- Une démonstration (utilisation) réelle, mais temporaire des services de l'application.
- Une interface de connexion
- Etc.

Inscription. L'application permet de s'inscrire selon trois profils :

- Parent : Dans ce cas, l'application lui propose :
 - o De saisir les informations du parent : nom, prénom, choix login, choix mot de passe, adresse mail, compte Tweeter, Facebook, etc. Permettre de vérifier l'adresse mail (Par exemple, activation du compte via un lien envoyé à l'adresse mail en renseignée).
 - o De choisir le nombre d'élèves concernés par cette inscription et de saisir leurs profils (nom, prénom, lien de parenté, niveau scolaire, etc.)
 - o De choisir l'année ou les années scolaires concernées.
 - o De choisir une formule parmi un ensemble de formules proposées. Par exemple :
 - § 1 cours par année scolaire.
 - § 2 cours par année scolaire.
 - § Tous les cours d'une année scolaire.
 - § Etc.

- o De choisir un mode de formation parmi deux :
 - § Formule progression : Documents électroniques uniquement.
 - § Formule accompagnement : accompagnement par un professeur (Accès à un tchat avec un enseignant pour poser ses questions dans toutes les matières.).
- o Propose un mode de paiement : carte bancaire, prélèvement, etc. et se saisir les informations nécessaires à ce mode de paiement.
- o Propose et envoie au parent (entre autres, par mail), des logins et mot de passe des élèves concernés. Les parents et les élèves peuvent ainsi se connecter en utilisant ces informations.
- Elève
 - o Même étape que « parent » avec la seule différence que l'élève sera enregistré sans parent.

Connexion. Selon le profil de connexion, les fonctionnalités offertes seront différentes :

- Parent
 - o Visualiser les moments de connexion de l'élève.
 - o Visualiser les cours et les exercices réalisés par l'élève.
 - o Visualiser les durées d'activité sur l'application.
 - o Visualiser des courbes de progression.
 - o Visualiser des recommandations pour la progression de l'élève.
 - § Recommandations pour une progression adaptée à l'élève concerné.
 - § Recommandations et conseils généraux.
 - o Définir des moments de rappel à l'élève pour revenir sur l'application.
 - o Etc.
- Élève
 - o Visualiser les différents articles de formation
 - § Cours
 - Vidéo
 - Fiches de cours
 - Présentation
 - § Exercices
 - QCM
 - Questions – Réponses
 - o Des cours en temps réel avec possibilité de pose de question.
 - Ces cours sont annoncés préalablement par notification, mails, etc.
 - o Tchat
 - § Des recommandations pour la progression de l'élève.
 - Recommandations pour une progression adaptée à l'élève concerné.
 - Recommandations et conseils généraux.
 - o Visualiser l'historique des activités sur l'application.
 - o Etc.

L'application permet deux modes de connexion : connexion online et connexion offline. La connexion online permet de se connecter au serveur et de bénéficier de tous les supports (cours, exercices, etc.) disponibles. En mode offline, l'utilisateur ne peut bénéficier que de certaines fonctionnalités en relation avec les données enregistrées localement. Une synchronisation doit être réalisée dès que l'utilisateur passe du mode offline au mode online.

Il est recommandé de réfléchir à la simplification des modes d'accès aux fonctionnalités de l'application :

- o Utilisation de certains capteurs (exemple, accélération) pour activer ou désactiver certaines fonctionnalités.
- o Utilisation de la voix pour activer ou désactiver certaines fonctionnalités.
- o Etc.

Partie serveur

La partie serveur doit permettre de :

- Stocker toutes les données nécessaires au fonctionnement de l'application, par exemple les comptes des utilisateurs, les cours sous leurs différents formats, etc.
- Implémenter certains algorithmes nécessaires au fonctionnement de l'application mobile.

Le choix de la technologie de réalisation de la partie serveur est libre. Néanmoins, veuillez à appliquer les meilleures pratiques pour cette partie importante à la majorité des applications mobiles.

Partie Android

En plus des services mentionnés ci-dessus, il est vivement recommandé d'utiliser les meilleures pratiques liées au développement Mobile (en Android) telles que Material Design, les fragments, les préférences, les content provider, etc.



Le projet MeFormer (ou MySchool) que tu proposes est une application mobile complexe qui doit permettre aux utilisateurs de suivre des cours en ligne et de gérer plusieurs profils (parents et élèves) avec différentes fonctionnalités comme la visualisation de cours, la gestion des progrès, etc. L'application devra être développée sous Android avec Kotlin, Jetpack Compose, et Firebase pour gérer les fonctionnalités de base telles que l'authentification et le stockage des données. Je vais te donner une approche structurée pour le développement de cette application en découpant les fonctionnalités principales.

1. Initialisation du Projet Android Studio avec Jetpack Compose

- Crée un projet Android dans Android Studio.
- Sélectionne **Jetpack Compose** comme framework UI.
- Ajoute Firebase à l'application (authentification et Firestore pour la gestion des données).

gradle

```
// build.gradle (Module)
implementation "com.google.firebase:firebase-auth-ktx"
implementation "com.google.firebase:firebase-firestore-ktx"
```

2. Présentation et Connexion

L'application doit présenter une interface simple pour les utilisateurs non-inscrits, leur permettant de naviguer et de tester certaines fonctionnalités.

- Crée une **page d'accueil** avec une présentation de l'application (texte, vidéo) et un bouton d'inscription ou de connexion.
- Implémente un **système d'authentification** via Firebase où les utilisateurs peuvent s'inscrire et se connecter.

kotlin

```
@Composable
fun WelcomeScreen(navController: NavController) {
    Column(modifier = Modifier.fillMaxSize().padding(16.dp), verticalArrangement =
Arrangement.Center) {
        Text(text = "Bienvenue dans MeFormer !", style = MaterialTheme.typography.h4)
        Button(onClick = { navController.navigate("login") }) {
            Text("Se connecter / S'inscrire")
        }
    }
}
```

3. Inscription selon le profil (Parent ou Élève)

L'utilisateur peut s'inscrire en tant que **Parent** ou **Élève** avec différentes informations à remplir selon le profil.

- Crée deux formulaires distincts pour l'inscription des parents et des élèves.
- Les parents doivent avoir la possibilité d'inscrire plusieurs enfants, et les élèves peuvent s'inscrire indépendamment.

Exemple pour un formulaire d'inscription des parents :

kotlin

```
@Composable
fun ParentRegistrationScreen(navController: NavController, viewModel: AuthViewModel) {
    var parentName by remember { mutableStateOf("") }
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp), verticalArrangement = Arrangement.Center) {
        TextField(value = parentName, onValueChange = { parentName = it }, label = {
Text("Nom du parent") })
        TextField(value = email, onValueChange = { email = it }, label = { Text("Email")
})
        TextField(value = password, onValueChange = { password = it }, label = { Text("Mot
de passe") }, visualTransformation = PasswordVisualTransformation())

        Button(onClick = { viewModel.registerParent(parentName, email, password) }) {
            Text("S'inscrire")
        }
    }
}
```

Utilise Firebase Authentication pour la création de compte et Firestore pour stocker les informations supplémentaires comme les profils des élèves liés.

4. Gestion des Cours et Contenus

Les utilisateurs peuvent accéder à des cours sous différents formats (vidéos, fiches de cours, QCM, etc.).

- Stocke les cours dans **Firebase Firestore** ou **Firebase Storage** (pour les vidéos et fichiers PDF).
- Crée une **interface utilisateur** permettant de visualiser les cours et d'accéder aux différents formats.
- Utilise des composants Jetpack Compose pour afficher des vidéos et du texte.

kotlin

```
@Composable
fun CourseScreen(courseId: String, viewModel: CourseViewModel) {
    val course by viewModel.getCourse(courseId).observeAsState()
    course?.let {
        Text(text = it.title, style = MaterialTheme.typography.h5)
        Text(text = it.description)
        Button(onClick = { /* Lancer la vidéo */ }) {
            Text("Voir la vidéo")
        }
    }
}
```

5. Mode Online et Offline

Pour gérer le mode offline, utilise **Room** pour stocker les données localement. Les utilisateurs doivent pouvoir accéder à certaines parties des cours et des exercices même sans connexion internet.

- Lors de la synchronisation avec Firebase, télécharge les données nécessaires (cours, exercices) pour une utilisation hors ligne.
- Implémente un système de notification pour prévenir l'utilisateur lorsqu'il passe en mode online/offline.

6. Gestion des Profils (Parent et Élève)

Après la connexion, les parents et élèves ont accès à des fonctionnalités différentes :

- **Parent :**
 - Voir les moments de connexion de l'élève.
 - Voir les progrès de l'élève sous forme de courbes.
 - Envoyer des rappels aux élèves.
- **Élève :**
 - Accéder aux cours, QCM, et exercices.
 - Suivre son historique d'activité.

Ces informations seront récupérées et mises à jour dans **Firebase Firestore**.

Exemple d'interface pour la visualisation des progrès :

kotlin

```
@Composable
fun ProgressScreen(viewModel: ProgressViewModel) {
    val progress by viewModel.getStudentProgress().observeAsState()
```

```
} LineChart(progress.data) // Graphique des progrès avec les données récupérées
```

7. Serveur et Stockage des Données

La partie serveur gère la base de données des utilisateurs, des cours, et des statistiques.

- Utilise **Firestore** pour stocker les données des utilisateurs et des cours.
- Pour les fichiers volumineux (vidéos, PDF), utilise **Storage**.

Voici un exemple pour la structure des documents dans Firestore :

- ``Users`` : contenant les informations des parents, élèves, et leurs profils.
- ``Courses`` : contenant les informations sur chaque cours (titre, description, fichiers associés).
- ``Progress`` : suivi des progrès des élèves (courbes, statistiques).

8. Fonctionnalités Avancées

- Intégrer des capteurs (ex. capteur d'accélération) pour activer ou désactiver des fonctionnalités.
- Utiliser la voix pour naviguer dans certaines parties de l'application.

Conclusion

Ce projet nécessite la gestion de plusieurs composants comme Firebase pour l'authentification et la gestion des données, Jetpack Compose pour l'interface utilisateur, et Room pour la gestion hors ligne. Tu peux également étendre les fonctionnalités en ajoutant plus de services pour les utilisateurs (chat en temps réel, notifications, etc.).



Modifie les dépendances de Firebase en utilisant le fichier `libs.version.toml` suivant :

[versions]

agp = "8.5.2"

kotlin = "1.9.0"

coreKtx = "1.13.1"

junit = "4.13.2"

junitVersion = "1.2.1"

espressoCore = "3.6.1"

appcompat = "1.7.0"

material = "1.12.0"

activity = "1.9.2"

constraintlayout = "2.1.4"

[libraries]

androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }

junit = { group = "junit", name = "junit", version.ref = "junit" }

androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }

androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", version.ref = "espressoCore" }

androidx-appcompat = { group = "androidx.appcompat", name = "appcompat", version.ref = "appcompat" }

```
material = { group = "com.google.android.material", name = "material", version.ref = "material" }
androidx-activity = { group = "androidx.activity", name = "activity", version.ref = "activity" }
androidx-constraintlayout = { group = "androidx.constraintlayout", name = "constraintlayout",
version.ref = "constraintlayout" }
```

```
[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
```



Voici comment intégrer Firebase dans ton projet Android en utilisant le fichier `libs.versions.toml` fourni et en modifiant les dépendances dans `build.gradle.kts` pour qu'elles soient compatibles avec Firebase tout en suivant la structure de la gestion des dépendances via le fichier `libs.versions.toml`.

Étape 1: Mise à jour du fichier `libs.versions.toml`

Ajoute les dépendances Firebase dans ton fichier `libs.versions.toml`. Cela permet de centraliser la gestion des versions.

toml

```
[versions]
agp = "8.5.2"
kotlin = "1.9.0"
coreKtx = "1.13.1"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
appcompat = "1.7.0"
material = "1.12.0"
activity = "1.9.2"
constraintlayout = "2.1.4"
firebase-bom = "32.2.0"
firebase-auth-ktx = "22.0.2"
firebase-firestore-ktx = "25.1.0"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", version.ref = "espressoCore" }
androidx-appcompat = { group = "androidx.appcompat", name = "appcompat", version.ref = "appcompat" }
material = { group = "com.google.android.material", name = "material", version.ref = "material" }
androidx-activity = { group = "androidx.activity", name = "activity", version.ref = "activity" }
androidx-constraintlayout = { group = "androidx.constraintlayout", name = "constraintlayout", version.ref = "constraintlayout" }

firebase-bom = { group = "com.google.firebase", name = "firebase-bom", version.ref = "firebase-bom" }
firebase-auth-ktx = { group = "com.google.firebase", name = "firebase-auth-ktx", version.ref = "firebase-auth-ktx" }
firebase-firestore-ktx = { group = "com.google.firebase", name = "firebase-firestore-ktx", version.ref = "firebase-firestore-ktx" }

[plugins]
```

```
android-application = { id = "com.android.application", version.ref = "agp" }
jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
```

Étape 2: Modifier `build.gradle.kts` (Module : app)

Dans le fichier `build.gradle.kts` de ton module `app`, tu vas intégrer Firebase à l'aide du BOM (Bill of Materials) de Firebase pour garantir la cohérence des versions.

kotlin

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

android {
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.meformer"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"),
"proguard-rules.pro")
        }
    }

    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.3"
    }

    kotlinOptions {
        jvmTarget = "1.9"
    }

    buildFeatures {
        compose = true
    }
}

dependencies {
    implementation(platform(libs.firebase.bom)) // Firebase BOM for consistent versioning
    implementation(libs.firebase.auth.ktx) // Firebase Authentication
    implementation(libs.firebase.firestore.ktx) // Firebase Firestore

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)

    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
}
```

Explications des dépendances :

1. **Firestore BOM** : Le BOM (`firebase-bom`) est utilisé pour gérer les versions de Firestore de manière cohérente. Il permet de synchroniser toutes les bibliothèques Firestore avec la même version sans avoir à les spécifier individuellement.
2. **Firestore Auth** : La bibliothèque Firestore pour gérer l'authentification.
3. **Firestore Firestore** : La bibliothèque Firestore pour gérer la base de données en temps réel.

Étape 3: Configurer Firestore dans le projet

1. Va sur [Firestore Console](#), crée un projet et ajoute ton application Android en récupérant le fichier `google-services.json`.
2. Ajoute ce fichier dans ton projet sous le répertoire `app/`.
3. Modifie ton fichier `build.gradle.kts` (niveau projet) pour inclure le plugin `google-services` :

```
kotlin

// build.gradle.kts (Project)
plugins {
    id("com.android.application") version "8.5.2" apply false
    id("com.android.library") version "8.5.2" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
    id("com.google.gms.google-services") version "4.3.15" apply false
}
```

Et dans le fichier `build.gradle.kts` (Module: app) ajoute le plugin Google Services en bas du fichier :

```
kotlin

apply(plugin = "com.google.gms.google-services")
```

Étape 4: Synchroniser le projet

Après avoir configuré toutes les dépendances, synchronise ton projet pour télécharger et intégrer les bibliothèques Firestore et Android.

Résumé

Voici les étapes pour intégrer Firestore avec la gestion des dépendances via le fichier `libs.versions.toml`. Tu as maintenant Firestore Auth et Firestore configurés pour ton application. Cela te permettra de gérer l'authentification des utilisateurs et le stockage des données dans Firestore tout en maintenant une gestion des versions centralisée avec le fichier `libs.versions.toml`.



Maintenant que les dépendances sont réalisées, on peut maintenant réaliser les différentes classes Kotlin avec Jetpack Compose de l'application MySchool



Bien sûr ! Passons maintenant à la création des différentes classes Kotlin pour l'application **MySchool** en utilisant Jetpack Compose. Voici une approche structurée pour le développement des composants principaux de l'application, incluant la navigation, l'authentification, l'inscription, la gestion des cours, et les écrans pour les parents et les élèves.

Structure du projet

Voici une suggestion pour la structure des fichiers de ton projet :

markdown

```
- com.example.myschool
  - MainActivity.kt
  - navigation
    - AppNavHost.kt
  - screens
    - LoginScreen.kt
    - RegistrationScreen.kt
    - ParentHomeScreen.kt
    - StudentHomeScreen.kt
    - CourseScreen.kt
    - ProfileScreen.kt
  - viewmodels
    - AuthViewModel.kt
    - CourseViewModel.kt
    - ProgressViewModel.kt
  - models
    - User.kt
    - Course.kt
    - Progress.kt
```

1. MainActivity.kt

C'est le point d'entrée de l'application où tu définis la configuration de base.

kotlin

```
package com.example.myschool

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import com.example.myschool.navigation.AppNavHost
import com.example.myschool.ui.theme.MySchoolTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MySchoolTheme {
                Surface(color = MaterialTheme.colors.background) {
                    AppNavHost()
                }
            }
        }
    }
}
```

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    MySchoolTheme {
        Text(text = "Hello, MySchool!")
    }
}
```

2. Navigation (AppNavHost.kt)

Configurer la navigation de l'application à l'aide de **Jetpack Navigation Compose**.

kotlin

```
package com.example.myschool.navigation

import androidx.compose.runtime.Composable
import androidx.navigation.NavHostController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import com.example.myschool.screens.*

@Composable
fun AppNavHost(navController: NavHostController) {
    NavHost(navController = navController, startDestination = "login") {
        composable("login") { LoginScreen(navController) }
        composable("registration") { RegistrationScreen(navController) }
        composable("parent_home") { ParentHomeScreen(navController) }
        composable("student_home") { StudentHomeScreen(navController) }
        composable("course/{courseId}") { backStackEntry ->
            val courseId = backStackEntry.arguments?.getString("courseId")
            CourseScreen(courseId ?: "", navController)
        }
        composable("profile") { ProfileScreen(navController) }
    }
}
```

3. Écrans

a. LoginScreen.kt

Écran de connexion pour les utilisateurs.

kotlin

```
package com.example.myschool.screens

import androidx.compose.foundation.layout.*
import androidx.compose.material.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.navigation.NavController
import com.example.myschool.viewmodels.AuthViewModel

@Composable
fun LoginScreen(navController: NavController, viewModel: AuthViewModel = AuthViewModel()) {
    {
        var email by remember { mutableStateOf("") }
        var password by remember { mutableStateOf("") }

        Column(
```

```

        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Se connecter", style = MaterialTheme.typography.h4)
        Spacer(modifier = Modifier.height(16.dp))
        TextField(value = email, onValueChange = { email = it }, label = { Text("Email")
    })

    Spacer(modifier = Modifier.height(8.dp))
    TextField(value = password, onValueChange = { password = it }, label = { Text("Mot
de passe") }, visualTransformation = PasswordVisualTransformation())
    Spacer(modifier = Modifier.height(16.dp))
    Button(onClick = { viewModel.login(email, password, navController) }) {
        Text("Se connecter")
    }
    Spacer(modifier = Modifier.height(8.dp))
    TextButton(onClick = { navController.navigate("registration") }) {
        Text("Créer un compte")
    }
}
}
}

```

b. RegistrationScreen.kt

Écran d'inscription pour les utilisateurs.

kotlin

```

package com.example.myschool.screens

import androidx.compose.foundation.layout.*
import androidx.compose.material.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.navigation.NavController
import com.example.myschool.viewmodels.AuthViewModel

@Composable
fun RegistrationScreen(navController: NavController, viewModel: AuthViewModel =
AuthViewModel()) {
    var name by remember { mutableStateOf("") }
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text("Créer un compte", style = MaterialTheme.typography.h4)
        Spacer(modifier = Modifier.height(16.dp))
        TextField(value = name, onValueChange = { name = it }, label = { Text("Nom") })
        Spacer(modifier = Modifier.height(8.dp))
        TextField(value = email, onValueChange = { email = it }, label = { Text("Email")
    })

    Spacer(modifier = Modifier.height(8.dp))
    TextField(value = password, onValueChange = { password = it }, label = { Text("Mot
de passe") }, visualTransformation = PasswordVisualTransformation())
    Spacer(modifier = Modifier.height(16.dp))
    Button(onClick = { viewModel.register(name, email, password, navController) }) {
        Text("S'inscrire")
    }
}
}

```

```
}
}
```

c. ParentHomeScreen.kt

Écran principal pour les parents.

kotlin

```
package com.example.myschool.screens

import androidx.compose.material.*
import androidx.compose.runtime.Composable
import androidx.navigation.NavController

@Composable
fun ParentHomeScreen(navController: NavController) {
    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Accueil Parent") })
        }
    ) {
        // Contenu de l'écran pour le parent
        Text("Bienvenue, Parent !", style = MaterialTheme.typography.h5)
        // Autres composants comme les cours, les statistiques des élèves, etc.
    }
}
```

d. StudentHomeScreen.kt

Écran principal pour les élèves.

kotlin

```
package com.example.myschool.screens

import androidx.compose.material.*
import androidx.compose.runtime.Composable
import androidx.navigation.NavController

@Composable
fun StudentHomeScreen(navController: NavController) {
    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Accueil Élève") })
        }
    ) {
        // Contenu de l'écran pour l'élève
        Text("Bienvenue, Élève !", style = MaterialTheme.typography.h5)
        // Autres composants comme les cours, les exercices, etc.
    }
}
```

e. CourseScreen.kt

Écran pour afficher un cours.

kotlin

```
package com.example.myschool.screens

import androidx.compose.material.*
import androidx.compose.runtime.Composable
import androidx.navigation.NavController

@Composable
fun CourseScreen(courseId: String, navController: NavController) {
    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Détails du Cours") })
        }
    ) {
        // Logique pour récupérer et afficher les détails du cours
        Text("Détails du cours pour l'ID : $courseId", style =
MaterialTheme.typography.h6)
        // Afficher la vidéo, les documents, les exercices, etc.
    }
}
```

f. ProfileScreen.kt

Écran pour gérer le profil de l'utilisateur.

kotlin

```
package com.example.myschool.screens

import androidx.compose.material.*
import androidx.compose.runtime.Composable
import androidx.navigation.NavController

@Composable
fun ProfileScreen(navController: NavController) {
    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Mon Profil") })
        }
    ) {
        // Logique pour afficher et gérer le profil de l'utilisateur
        Text("Gérer mon profil", style = MaterialTheme.typography.h6)
    }
}
```

4. ViewModels

a. AuthViewModel.kt

ViewModel pour gérer l'authentification des utilisateurs.

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.ViewModel
import androidx.navigation.NavController
import com.google.firebase.auth.FirebaseAuth

class AuthViewModel : ViewModel() {
```

```
private val auth: FirebaseAuth = FirebaseAuth.getInstance()

fun login(email: String, password: String, navController: NavController) {
    auth.signInWithEmailAndPassword(email, password).addOnCompleteListener { task ->
        if (task.isSuccessful) {
            navController.navigate("parent_home") // Ou "student_home" selon le profil
        } else {
            // Gérer l'erreur
        }
    }
}

fun register(name: String, email: String, password: String, navController:
NavController) {
    auth.createUserWithEmailAndPassword(email, password).addOnCompleteListener { task
->
        if (task.isSuccessful) {
            // Ajouter l'utilisateur à la base de données
            navController.navigate("login")
        } else {
            // Gérer l'erreur
        }
    }
}
}
```

b. CourseViewModel.kt

ViewModel pour gérer les cours.

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.ViewModel

class CourseViewModel : ViewModel() {
    // Logique pour gérer les cours, récupérer les données, etc.
}
```

c. ProgressViewModel.kt

ViewModel pour gérer le progrès des élèves.

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.ViewModel

class ProgressViewModel : ViewModel() {
    // Logique pour gérer le progrès de l'élève, récupérer les notes, etc.
}
```

5. Modèles

a. User.kt

Modèle pour représenter un utilisateur.

kotlin

```
package com.example.myschool.models

data class User(
    val id: String,
    val name: String,
    val email: String,
    val userType: UserType // Enum pour Parent ou Élève
)

enum class UserType {
    PARENT,
    STUDENT
}
```

b. Course.kt

Modèle pour représenter un cours.

kotlin

```
package com.example.myschool.models

data class Course(
    val id: String,
    val title: String,
    val description: String,
    val teacherId: String
)
```

c. Progress.kt

Modèle pour représenter le progrès de l'élève.

kotlin

```
package com.example.myschool.models

data class Progress(
    val courseId: String,
    val userId: String,
    val score: Double
)
```

Conclusion

Avec cette structure de base, tu as mis en place les écrans essentiels et les ViewModels nécessaires pour faire fonctionner l'application **MySchool**. Bien sûr, tu peux enrichir ces classes avec davantage de fonctionnalités et d'interactions selon les besoins de ton application, telles que la gestion des erreurs, la connexion à Firestore pour stocker et récupérer des données, et l'ajout de fonctionnalités avancées comme la messagerie, les notifications, etc.

N'hésite pas à demander si tu souhaites approfondir un aspect spécifique ou ajouter des fonctionnalités supplémentaires !

Je souhaite réaliser un thème `MySchoolTheme` utilisé dans toute l'application afin de corriger les erreurs de la classe `MainActivity`



Pour créer un thème personnalisé dans une application utilisant Jetpack Compose, tu dois définir une classe de thème qui contient les couleurs, les typographies et les styles que tu souhaites utiliser dans toute l'application. Je vais te montrer comment créer un thème `MySchoolTheme` pour ton application **MySchool**.

Étape 1 : Créer les Couleurs

Tout d'abord, définis les couleurs que tu souhaites utiliser dans ton thème. Crée un fichier `Color.kt` dans un dossier `ui/theme` :

kotlin

```
package com.example.myschool.ui.theme

import androidx.compose.ui.graphics.Color

val PrimaryColor = Color(0xFF6200EE) // Change cette couleur selon ton choix
val PrimaryVariantColor = Color(0xFF3700B3)
val SecondaryColor = Color(0xFF03DAC5)

val BackgroundColor = Color(0xFFFFFFFF) // Blanc
val SurfaceColor = Color(0xFFEEEEEE) // Gris clair
val ErrorColor = Color(0xFFB00020)

val OnPrimaryColor = Color.White
val OnSecondaryColor = Color.Black
val OnBackgroundColor = Color.Black
val OnSurfaceColor = Color.Black
val OnErrorColor = Color.White
```

Étape 2 : Créer la Typographie

Ensuite, définis la typographie que tu souhaites utiliser dans ton thème. Crée un fichier `Typography.kt` dans le même dossier :

kotlin

```
package com.example.myschool.ui.theme

import androidx.compose.material.Typography
import androidx.compose.material.TextStyle
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp

val MyTypography = Typography(
    h1 = TextStyle(
        fontSize = 30.sp,
        fontWeight = FontWeight.Bold
    ),
    h2 = TextStyle(
        fontSize = 24.sp,
        fontWeight = FontWeight.Bold
    )
)
```

```

    ),
    body1 = TextStyle(
        fontSize = 16.sp,
        fontWeight = FontWeight.Normal
    ),
    button = TextStyle(
        fontSize = 14.sp,
        fontWeight = FontWeight.Bold
    )
)

```

Étape 3 : Créer le Thème

Maintenant, crée le fichier `Theme.kt` dans le même dossier pour assembler tout cela :

kotlin

```

package com.example.myschool.ui.theme

import androidx.compose.material.MaterialTheme
import androidx.compose.runtime.Composable
import androidx.compose.runtime.Stable
import androidx.compose.ui.graphics.Color

@Stable
class MySchoolTheme(
    val colors: MyColors,
    val typography: MyTypography
)

@Composable
fun MySchoolTheme(content: @Composable () -> Unit) {
    MaterialTheme(
        colors = MaterialTheme.colors.copy(
            primary = PrimaryColor,
            primaryVariant = PrimaryVariantColor,
            secondary = SecondaryColor,
            background = BackgroundColor,
            surface = SurfaceColor,
            error = ErrorColor,
            onPrimary = OnPrimaryColor,
            onSecondary = OnSecondaryColor,
            onBackground = OnBackgroundColor,
            onSurface = OnSurfaceColor,
            onError = OnErrorColor
        ),
        typography = MyTypography,
        shapes = MaterialTheme.shapes,
        content = content
    )
}

```

Étape 4 : Modifier la `MainActivity.kt`

Ensuite, utilise ce thème dans ta `MainActivity`. Modifie la méthode `setContent` comme suit :

kotlin

```

package com.example.myschool

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import com.example.myschool.navigation.AppNavHost

```

```
import com.example.myschool.ui.theme.MySchoolTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MySchoolTheme { // Utilisation de MySchoolTheme
                AppNavHost()
            }
        }
    }
}
```

Étape 5 : Utiliser le Thème dans les Écrans

Maintenant, tu peux utiliser les styles définis dans ton thème dans tous tes écrans. Par exemple, dans `LoginScreen.kt`, tu peux utiliser le style de texte que tu as défini dans `MyTypography` :

kotlin

```
@Composable
fun LoginScreen(navController: NavController, viewModel: AuthViewModel = AuthViewModel())
{
    // ...
    Text("Se connecter", style = MaterialTheme.typography.h4) // Utilise la typographie
    définie
    // ...
}
```

Conclusion

Avec ces étapes, tu as maintenant un thème global pour ton application **MySchool** qui garantit une cohérence dans le design de l'interface. Tu peux personnaliser les couleurs et la typographie selon tes préférences. N'hésite pas à ajuster les valeurs des couleurs et des styles selon tes besoins. Si tu as d'autres questions ou souhaites explorer d'autres fonctionnalités, fais-le moi savoir !



Donne-moi le code des classes ViewModels que tu ne m'as pas fourni



Bien sûr ! Voici les classes `ViewModel` que tu peux utiliser dans ton application **MySchool**. Ces classes gèrent la logique pour les fonctionnalités de ton application, telles que l'authentification des utilisateurs, la gestion des cours, et le suivi des progrès des élèves.

1. AuthViewModel.kt

Le `AuthViewModel` gère l'authentification des utilisateurs (connexion et inscription).

kotlin

```
package com.example.myschool.viewmodels
```

```
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.google.firebase.auth.FirebaseAuth
import kotlinx.coroutines.launch

class AuthViewModel : ViewModel() {
    private val auth: FirebaseAuth = FirebaseAuth.getInstance()

    // Fonction pour connecter un utilisateur
    fun login(email: String, password: String, onSuccess: () -> Unit, onFailure: (String)
-> Unit) {
        viewModelScope.launch {
            auth.signInWithEmailAndPassword(email, password)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        onSuccess()
                    } else {
                        onFailure(task.exception?.message ?: "Erreur inconnue")
                    }
                }
        }
    }

    // Fonction pour enregistrer un nouvel utilisateur
    fun register(email: String, password: String, onSuccess: () -> Unit, onFailure:
(String) -> Unit) {
        viewModelScope.launch {
            auth.createUserWithEmailAndPassword(email, password)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        onSuccess()
                    } else {
                        onFailure(task.exception?.message ?: "Erreur inconnue")
                    }
                }
        }
    }

    // Fonction pour se déconnecter
    fun logout() {
        auth.signOut()
    }

    // Vérifier si l'utilisateur est connecté
    fun isUserLoggedIn(): Boolean {
        return auth.currentUser != null
    }
}
```

2. CourseViewModel.kt

Le `CourseViewModel` gère les opérations liées aux cours, comme la récupération des données des cours depuis Firestore.

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.example.myschool.models.Course
import com.google.firebase.firestore.FirebaseFirestore

class CourseViewModel : ViewModel() {
    private val db: FirebaseFirestore = FirebaseFirestore.getInstance()

    private val _courses = MutableLiveData<List<Course>>()
    val courses: LiveData<List<Course>> get() = _courses
}
```

```
// Fonction pour récupérer les cours depuis Firestore
fun fetchCourses() {
    db.collection("courses")
        .get()
        .addOnSuccessListener { result ->
            val courseList = result.map { document ->
                Course(
                    id = document.id,
                    title = document.getString("title") ?: "",
                    description = document.getString("description") ?: "",
                    teacherId = document.getString("teacherId") ?: ""
                )
            }
            _courses.value = courseList
        }
        .addOnFailureListener { exception ->
            // Gérer l'erreur ici
            _courses.value = emptyList() // Ou gérer l'erreur de manière appropriée
        }
}
}
```

3. ProgressViewModel.kt

Le `ProgressViewModel` gère le suivi des progrès des élèves.

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import com.example.myschool.models.Progress
import com.google.firebase.firestore.FirebaseFirestore

class ProgressViewModel : ViewModel() {
    private val db: FirebaseFirestore = FirebaseFirestore.getInstance()

    private val _progressList = MutableLiveData<List<Progress>>()
    val progressList: LiveData<List<Progress>> get() = _progressList

    // Fonction pour récupérer le progrès d'un élève depuis Firestore
    fun fetchProgress(userId: String) {
        db.collection("progress")
            .whereEqualTo("userId", userId)
            .get()
            .addOnSuccessListener { result ->
                val progressData = result.map { document ->
                    Progress(
                        courseId = document.getString("courseId") ?: "",
                        userId = document.getString("userId") ?: "",
                        score = document.getDouble("score") ?: 0.0
                    )
                }
                _progressList.value = progressData
            }
            .addOnFailureListener { exception ->
                // Gérer l'erreur ici
                _progressList.value = emptyList() // Ou gérer l'erreur de manière
                appropriée
            }
    }
}
```

Conclusion

Ces `ViewModels` fournissent la logique nécessaire pour gérer l'authentification, les cours et le suivi des progrès des élèves dans ton application **MySchool**. Tu peux les étendre en ajoutant des méthodes supplémentaires ou en améliorant la gestion des erreurs selon tes besoins.

N'hésite pas à me faire savoir si tu as besoin d'autres ajouts ou de précisions sur un point particulier !



J'ai un problème de type dans la classe AuthViewModel où le type NavController est envoyé à la méthode login qui demande un type Unit.



Il semble que tu rencontres une confusion dans la gestion des paramètres dans la méthode `login` de ta classe `AuthViewModel`. Pour clarifier, voici une version corrigée de la méthode `login`, ainsi que la façon de l'appeler dans ta logique d'interface utilisateur.

Correction de la méthode `login`

Dans la classe `AuthViewModel`, assure-toi que la méthode `login` n'attend que les paramètres nécessaires (email et password), et utilise des callbacks pour les résultats. Voici comment tu peux organiser cela :

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.google.firebase.auth.FirebaseAuth
import kotlinx.coroutines.launch

class AuthViewModel : ViewModel() {
    private val auth: FirebaseAuth = FirebaseAuth.getInstance()

    // Fonction pour connecter un utilisateur
    fun login(
        email: String,
        password: String,
        onSuccess: () -> Unit,
        onFailure: (String) -> Unit
    ) {
        viewModelScope.launch {
            auth.signInWithEmailAndPassword(email, password)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        onSuccess() // Callback pour indiquer le succès
                    } else {
                        onFailure(task.exception?.message ?: "Erreur inconnue") //
                        Callback pour indiquer l'échec
                    }
                }
        }
    }

    // ... (autres méthodes)
}
```

Appel de la méthode `login` dans l'interface utilisateur

Lorsque tu appelles la méthode `login` depuis ton interface utilisateur (par exemple, dans un écran de connexion), tu dois passer les fonctions `onSuccess` et `onFailure` qui gèrent respectivement le succès et l'échec de la connexion.

Voici un exemple de comment cela pourrait se présenter dans une `LoginScreen` :

kotlin

```
@Composable
fun LoginScreen(navController: NavController, viewModel: AuthViewModel = AuthViewModel())
{
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }
    var errorMessage by remember { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp)) {
        TextField(
            value = email,
            onValueChange = { email = it },
            label = { Text("Email") }
        )
        TextField(
            value = password,
            onValueChange = { password = it },
            label = { Text("Mot de passe") },
            visualTransformation = PasswordVisualTransformation()
        )
        Button(
            onClick = {
                viewModel.login(email, password,
                    onSuccess = {
                        // Naviguer vers l'écran principal après une connexion réussie
                        navController.navigate("home")
                    },
                    onFailure = { message ->
                        // Gérer l'affichage de l'erreur
                        errorMessage = message
                    }
                )
            }
        ) {
            Text("Se connecter")
        }

        if (errorMessage.isNotEmpty()) {
            Text(text = errorMessage, color = Color.Red)
        }
    }
}
```

Résumé

- **Correction dans `AuthViewModel` :** La méthode `login` prend maintenant des callbacks pour gérer le succès et l'échec.
- **Appel dans l'interface utilisateur :** Lors de l'appel à `login`, tu fournis des fonctions pour gérer la navigation et l'affichage des messages d'erreur.

Cela devrait résoudre ton problème de type et améliorer l'organisation de ton code. Si tu as besoin de plus d'aide ou d'autres clarifications, fais-le moi savoir !



Il y a le même problème pour la méthode `register` de `AuthViewModel`



Pour résoudre le même problème que tu as rencontré avec la méthode `login` dans la méthode `register` de la classe `AuthViewModel`, tu peux suivre une approche similaire. Assure-toi que la méthode `register` attend des paramètres appropriés et utilise des callbacks pour indiquer le succès ou l'échec de l'inscription.

Mise à jour de la méthode `register`

Voici comment tu peux structurer la méthode `register` dans `AuthViewModel` :

kotlin

```
package com.example.myschool.viewmodels

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.google.firebase.auth.FirebaseAuth
import kotlinx.coroutines.launch

class AuthViewModel : ViewModel() {
    private val auth: FirebaseAuth = FirebaseAuth.getInstance()

    // Fonction pour enregistrer un nouvel utilisateur
    fun register(
        email: String,
        password: String,
        onSuccess: () -> Unit,
        onFailure: (String) -> Unit
    ) {
        viewModelScope.launch {
            auth.createUserWithEmailAndPassword(email, password)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        onSuccess() // Callback pour indiquer le succès
                    } else {
                        onFailure(task.exception?.message ?: "Erreur inconnue") //
                        Callback pour indiquer l'échec
                    }
                }
        }
    }

    // ... (autres méthodes comme login, logout, etc.)
}
```

Appel de la méthode `register` dans l'interface utilisateur

Voici comment tu pourrais appeler la méthode `register` depuis une interface utilisateur, par exemple dans une `RegisterScreen` :

kotlin

```
@Composable
fun RegisterScreen(navController: NavController, viewModel: AuthViewModel =
    AuthViewModel()) {
    var email by remember { mutableStateOf("") }
}
```



```

var password by remember { mutableStateOf("") }
var errorMessage by remember { mutableStateOf("") }

Column(modifier = Modifier.padding(16.dp)) {
    TextField(
        value = email,
        onValueChange = { email = it },
        label = { Text("Email") }
    )
    TextField(
        value = password,
        onValueChange = { password = it },
        label = { Text("Mot de passe") },
        visualTransformation = PasswordVisualTransformation()
    )
    Button(
        onClick = {
            viewModel.register(email, password,
                onSuccess = {
                    // Naviguer vers l'écran principal ou un écran de confirmation
                    navController.navigate("home")
                },
                onFailure = { message ->
                    // Gérer l'affichage de l'erreur
                    errorMessage = message
                }
            )
        }
    ) {
        Text("S'inscrire")
    }

    if (errorMessage.isNotEmpty()) {
        Text(text = errorMessage, color = Color.Red)
    }
}

```

Résumé des modifications

- **Callback dans `register`** : La méthode `register` dans `AuthViewModel` a été mise à jour pour accepter des callbacks `onSuccess` et `onFailure`, permettant une gestion appropriée des résultats.
- **Appel dans l'interface utilisateur** : Lors de l'appel de `register`, tu fournis les fonctions pour naviguer ou afficher les erreurs.

Cela devrait résoudre les problèmes de type que tu avais avec la méthode `register`. Si tu as besoin de plus d'aide ou d'autres clarifications, n'hésite pas à demander !

ChatGPT peut faire des erreurs. Envisagez de vérifier les informations importantes.