Go4Lunch

TROUVEZ UN RESTAURANT POUR DEJEUNER AVEC VOS COLLEGUES

Guillaume Toussaint – Juillet 2021
OPEN CLASS ROOMS – DEVELOPPEUR D'APPLICATION ANDROID

Documentation technique

Présentation de l'application

« L'application Go4Lunch est une application collaborative utilisée par tous les employés. Elle permet de rechercher un restaurant dans les environs, puis de sélectionner celui de son choix en en faisant part à ses collègues. De la même manière, il est possible de consulter les restaurants sélectionnés par les collègues afin de se joindre à eux. Un peu avant l'heure du déjeuner, l'application notifie les différents employés pour les inviter à rejoindre leurs collègues. »

L'application a été développée pour Android avec Android Studio.
Le langage utilisé est le Java.
Elle supporte android 4.4 Kitkat

Git

Les sources de l'application sont hébergées sur Github

https://github.com/gtdevgit/P7

Fournisseurs de service tiers

Pour fonctionner l'application utilise des services tiers, elle est centrée sur une solution Google qui s'appelle Firebase. On utilise également les comptes développeur de Facebook et de Twitter et Google places pour obtenir des données géolocalisées.

Firebase

Firebase est une solution cloud proposer par Google pour réaliser des back-end à destination des applications mobiles. Firebase dispose d'un SDK pour Android. Dans ce projet nous utilisons *Firebase* pour ses services d'*authentification (auth)* et de *base de données (Firestore)*.

Pour utiliser Firebase il faut

- Un projet Android
  - Cible API 16 ou supérieur
  - Utilise Gradle 4.1 ou supérieur
  - Utilise Jetpack (Androidx) compliSdkVersion 28 ou supérieur
  - Nanespace
- Un compte Google
  - Créer un projet Firebase
  - Créer une application
  - Configurer la facturation
  - Configurer les autorisations liées aux services google
    - Service d'identifications (google, facebook, Twitter)
    - Service d'API GoogleMap, GooglePlace
    - Créer une clé d'API
- Configurer le projet Android
  - Générer et importer le fichier google-services-json

https://console.firebase.google.com/u/0/
https://firebase.google.com/docs/android/setup?authuser=0

Authentification (firebase)

L'authentification utilise le SDK Firebase et FirebaseUI

```
com.google.firebase.auth
com.firebase.ui.auth
```

https://firebase.google.com/docs/auth/android/firebaseui?authuser=0

Fournisseurs de connexion

Identification avec Google, Facebook, et Twitter.

Activer les différents types d'identifications voulus dans la console Firestore. Onglet Sign-in method.

Créer des comptes développeurs et paramétrer les applications pour chaque fournisseur de service.

- Google Compte https://myaccount.google.com/
- Facebook For Developers https://developers.facebook.com/apps
- Twitter Developer Portal https://developer.twitter.com/en/portal/projects-and-apps

Firestore Database

Firestore est une *base de données no SQL* disponible dans Firebase.

https://firebase.google.com/docs/firestore?authuser=0

Google Maps

Un widget permettant d'afficher une carte géographique. Vient de *Google Maps SDK*. On utilise la classe GoogleMap du pakage com.google.android.gms.maps pour manipuler la carte.

```
com.google.android.gms:play-services-maps:17.0.1
```

https://developers.google.com/android/reference/com/google/android/gms/maps/package-summary

Google Place API

« L'API Places est un service qui retourne des informations sur les lieux à l'aide de requêtes http »

https://developers.google.com/maps/documentation/places/web-service/overview

Pour pouvoir utiliser cette API il faut activer l'api dans la console Google et effectuer les requêtes avec la *clé d'API*. Cf. google-service.json.

- *Postman* permet de se familiariser avec l'API.
- Les échanges http se feront avec la librairie *Retrofit*.
- *OkHttpClient* permet d'ajouter des traces pour le débogage de l'application.
- Les retours JSON seront sérialisés avec la librairie Gson (GsonConverterFactory).

GPS

L'application utilise le GPS pour localiser l'utilisateur. Pour cela elle utilise la classe *FusedLocationProviderClient* du package com.google.android.gms.location et La classe *PermissionChecker* vérifiera que cette permission est accordée.

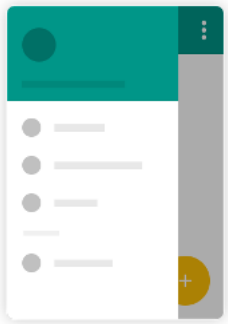
https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient

Cela nécessite la permission ACCESS_FINE_LOCATION qui sera ajouté dans le fichier AndroidManifest.xml de l'application.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Projet

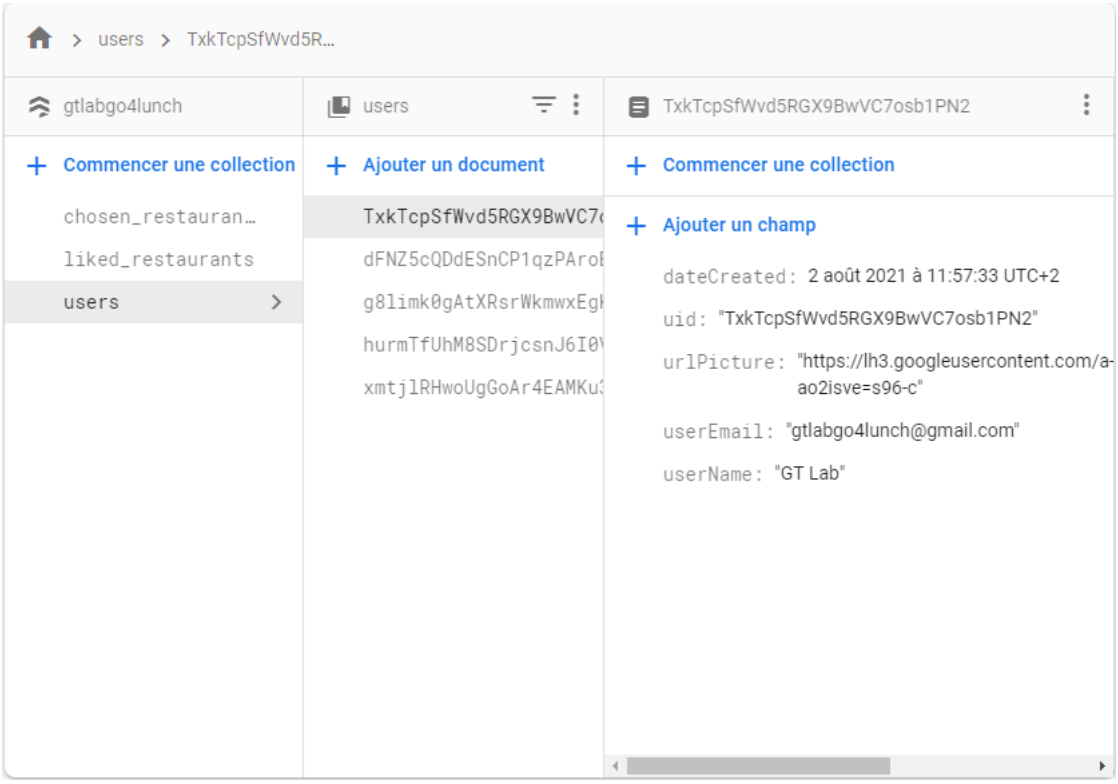
Le projet est construit sur le template « Navigation Drawer Activity » qui correspond à la demande exprimée par le cahier des charges.



Navigation Drawer Activity

Base de données

La base de données est hébergée sur Firebase – Firestore Database. C'est une base de données no SQL qui a la structure suivante :



Structure de la base de données

```
collection : users
[
        document : key = uid firebase
        {
                dateCreated : date
                uid : string
                urlPicture : string
                userEmail : string
                userName : string
        }
]

collection : liked_restaurants
[
        document : key = uid_placeid
        {
                createdTime : timestamp
                placeId : string
                userUid : string
```

```
        }
]

collection : chosen_restaurants
[
        document : key = uid
        {
                createdTime : timestamp
                placeId : string
                userUid : string
        }
]
```

Règles de gestion et de cardinalité

Liked :

Un restaurant peut être « liker » par plusieurs utilisateurs
Un utilisateur peut « liker » plusieurs restaurants
Un « like » est une relation d'unicité entre un utilisateur et un restaurant (un utilisateur ne peut pas cumuler les « like » sur le même restaurant)

Chosen :

Un utilisateur ne peut choisir qu'un seul restaurant à la fois.
Un restaurant peut être choisi par plusieurs utilisateurs.

Architecture

Android Architecture Components

L'application utilise les éléments logiciel fournis par *Android Architecture Components*. Cela permet de mettre en place une architecture MVVM.

https://developer.android.com/topic/libraries/architecture

Patron de conception MVVM Model, View, ViewModel

L'architecture générale de l'application met en œuvre le pattern MVVM.
Classes *ViewModel, ViewModelProvider.Factory, MutableLiveData, LiveData, Transformation, MediatorLiveData*.

Patron de conception « singleton »

Utilisé par les fabriques des *ViewModel*.

Patron de conception « observer »

La communication entre les couches applicatives est réalisée avec le pattern « Observer ». On utilise pour cela les classes *MutableLiveData*, *LiveData* et *MeditorLiveData*.

MainApplication

MainApplication est une classe de service pour l'application.

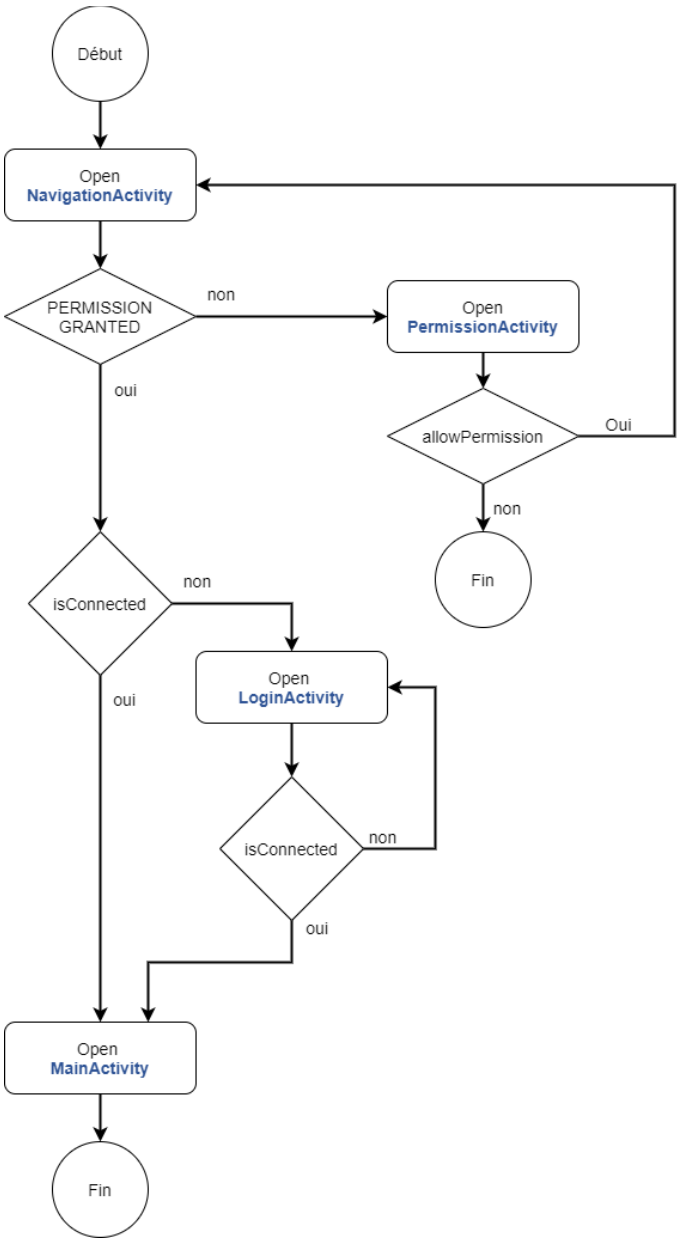
- Elle mets à disposition de l'ensemble de l'application l'objet Application, permettant d'accéder au contexte de l'application et ressources : getApplication()
- Elle expose également la clé de l'API Google : getGoogleApiKey()
- Elle crée le canal des notifications : createNotificationChannels(this)

Package navigation

Ce package contient l'activité *NavigationActivity*.

C'est une activité sans vue qui sert à définir lors du lancement de l'application l'activité à afficher. Elle utilise le processus ci-dessous pour déterminer quelle activité afficher.

Processus d'ouverture de l'application



Le package data

Le package « data » correspond à la couche Model de l'application. Il est divisé en sous-package, chaque package correspondant à un type de service.

Dans ces packages essentiellement on va trouver :

- Les classes POJO pour le transport et la sérialisation des données.
- Les repository pour le chargement et la mise à jour des données.

data.applicationsettings

Accès au setting de l'application, pour l'enregistrement du choix des notifications.
Définie *SettingRepository*
Utilise sharedpreference.

data.firestore

Accès à la base de données Firestore.
Utilise le SDK com.google.firebase.firestore.

Repository : *FirestoreChosenRepository, FirestoreLikedRepository, FirestoreUsersRepository*
Model : *User, UidPlaceIdAssociation*
Callback : *FailureListener, UserListListener, UserRestaurantAssociationListListener*
Validation : *CurrentTimeLimits*

data.googleplace

Accès à l'API Google Places
Nécessite la clé d'API passée en paramètre lors de l'instanciation du repository.
Retrofit pour les échanges http
GsonConverterFactory pour la sérialisation des retour JSON
OkHttpClient pour le débogage

Repository : *GooglePlacesApiRepository*
Api : *GooglePlacesApiClient*
Interface : *GooglePlacesApiInterface*
Model : *autocomplete, placesearch et placedetails*

data.permission_checker

Package de service pour vérifier l'autorisation de localisation.
Classe *PermissionChecker*
Utilise *ContextCompat.checkSelfPermission*

data.location

Accès à la localisation par GPS.
Intervalle : 10 minutes
Déplacement : 55 mètres

Utilise : *FusedLocationProviderClient*
Repository : LocationRepository

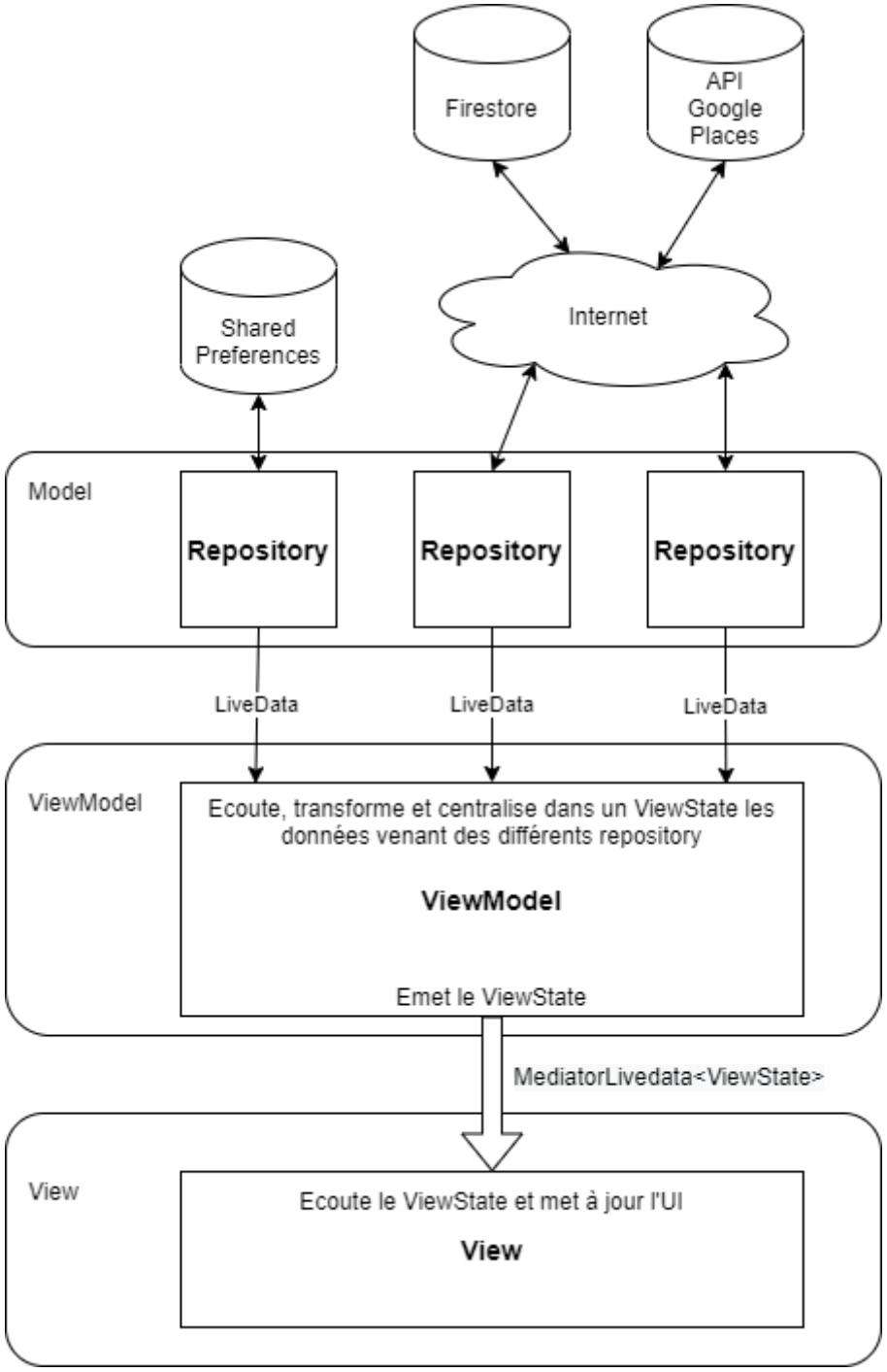
Package ui

Le package ui regroupe les activités et fragments ayant une vue.
Il est divisé en sous-package, chaque package correspondant à une ou plusieurs vues.

Chaque package est organisé de façon à contenir

- Un ViewModel
    - Responsable de la communication avec les repository, centralise l'acquisition, update et traitement des données.
    - Responsable de la communication avec la vue. Expose un ViewState.
- Un ViewModel factory
    - Factory pour le ViewModel
- Un ViewSate
    - Model spécialisé pour la vue. C'est un « container » qui véhicule les données entre le ViewModel et la vue.
- Une vue
    - La vue « écoute » le ViewState et se met a jour quand il change.
    - Affichage des données
    - Interaction avec l'utilisateur

Schéma MVVM


Firestore
API Google Places
Shared Preferences
Internet
Model
Repository
Repository
Repository
LiveData
LiveData
LiveData
ViewModel
Ecoute, transforme et centralise dans un ViewState les données venant des différents repository
ViewModel
Emet le ViewState
MediatorLivedata<ViewState>
View
Ecoute le ViewState et met à jour l'UI
View

Main et home

MainActivity est la vue principale, elle est divisée en 3 vues :

- Map
- ListView
- Workmate

Chaque vue est hébergée par un fragment. Ces vues utilisent les mêmes données, aussi elles partagent le même ViewModel, le découpage est fait de la façon suivante :

Package main

Ce package contient le *MainViewModel , MainViewModelFactory, MainViewState, MainActivity*

Navigation

La navigation entre les fragments *home, setting et logout* du navigation_drawer est gérer par la MainActivity qui acquière les fragments disponibles depuis la ressources res/navigation/mobile_navigation.xml pour les associer avec le NavigationView. Cela est effectué dans la procédure onCreate()

Package home

Ce package contient les fragments pour les vues *map, listview et workmates*

Package detailrestaurant

La vue DetailRestaurantActivity

Package loginActivity

La vue LoginActivity

Pacakge logout

Le fragment logoutFragment

Package setting

Le fragment SettingFragment

Package notification

Ce package est responsable de la création et de la planification des notifications.

NotificationHelper

Cette classe propose un helper pour gérer les notifications.

- *createNotificationChanenels* : Permet à l'application de créer un canal pour ses notifications. Est appelé par Mainapplication.
- *startNotificationWorker*
- *stopNotification*
- *sendNotification* : Les paramètres
- *createMessage* : Interroge les repository pour créer le message qui sera inclue dans la notification.

NotificationWorker

Cette classe hérite de *Worker*. La procédure *doWork* sera exécuté lorsque que la notification se produira.

- *doWork*()
    - Appelle NotificationHelper.*createMessage* qui créera le contenue de la notification.
    - La notification est émise par le listener NotificationMessageListener avec NotificationHelper.*sendNotification*.
    - Appelle NotificationHelper.*startNotificationWorker* pour relancer la prochaine notification à H+24.

Tests Unitaires

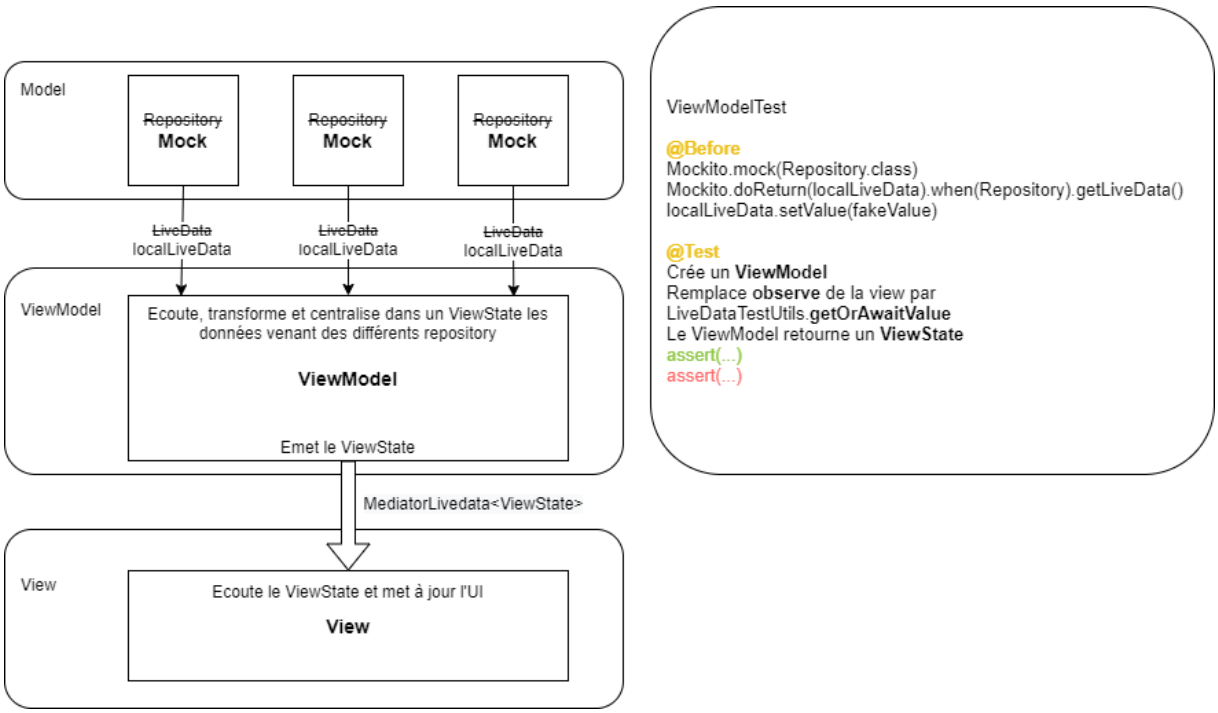
Les tests unitaires testent les classes POJO même si cela a son utilité cela reste d'un intérêt limité car ces tests se résument à contrôler des constructeurs et des getter. Dans une architecture MVVM il est plutôt bénéfique de tester la logique du ViewModel.

Le principe de test unitaires avec les larchitecture MVVM d'Android consiste à tester les ViewModel en « mockant » les repository et à remplacer les valeurs issues des repository par des valeurs fictives puis à comparer ce qui est retourné dans le ViewState avec des valeurs théoriques.

C'est ce qui est fait pour le MainViewModel et le DetailRestaurantViewModelTest

```
com.exemple.go4lunch.ui.main.viewmodel.MainViewModelTest
com.exemple.go4lunch.ui.detailrestaurant.viemodel.DetailRestaurantViewModelTest
```

On utilise pour cela la librairie Mockito.



Remarque : L'instanciation d'un ViewModel lors du processus de test est rendue possible car les ViewModel demande les repository en paramètre de leur constructeur (injection de dépendance).