

STOCKAGE DE DONNÉES AVEC LA LIBRAIRIE ROOM



CONTENU

- Composantes architecturales
- Entités
- DAO
- Base de données (Room)
- ViewModel
- Repository
- LiveData
- Lifecycle

COMPOSANTS D'ARCHITECTURE

COMPOSANTS D'ARCHITECTURE

Un ensemble de bibliothèques vous permettant de structurer vos applications avec robustesse, de tester plus facilement vos applications, garantissant la maintenance, etc.



COMPOSANTS D'ARCHITECTURE

- Ensemble de Meilleures pratiques architecturales + librairies
- Architecture recommandée par Google
- MOINS DE répétitions de code
- Testable / séparation claire
- Moins de dépendances
- Maintenance facile

GUIDE ARCHITECTURAL

The screenshot shows the 'Guide to App Architecture' page on the Android Developer website. The page has a green header with 'Developers' and navigation links for 'DESIGN', 'DEVELOP', and 'DISTRIBUTE'. A search bar is also present. On the left, a sidebar lists various libraries, with 'Architecture Components' expanded to show 'Guide to App Architecture'. The main content area is titled 'Guide to App Architecture' and includes an introduction, a note about the guide's assumptions, and a section on 'Common problems faced by app developers'. To the right of the main text, there are links to the 'Issue Tracker' and 'G+ Community'. At the bottom right, a box titled 'In this document' lists links to specific sections of the guide.

Guide to App Architecture | Android Developers

Secure | <https://developer.android.com/topic/libraries/architecture/guide.html>

Developers

DESIGN DEVELOP DISTRIBUTE

Search

Libraries

Architecture Components

[Guide to App Architecture](#)

Adding Components to your Project

Handling Lifecycles

LiveData

ViewModel

Room Persistence Library

Paging Library

Feedback

Release Notes

Support Library

Data Binding Library

Testing Support Library

Play Billing Library

Develop > Libraries > Architecture Components

Guide to App Architecture

This guide is for developers who are past the basics of building an app, and now want to know the best practices and recommended architecture for building robust, production-quality apps.

Note: This guide assumes that the reader has familiarity with the Android Framework. If you are new to app development, check out the [Getting Started](#) training series, which covers prerequisite topics for this guide.

Common problems faced by app developers

Unlike their traditional desktop counterparts which, in the majority of cases, have a single entry point from the launcher shortcut and run as a single monolithic process, Android apps have a much more complex structure. A typical Android app is constructed out of multiple [app components](#), including activities, fragments, services, content providers and broadcast receivers.

> **Issue Tracker**

Report issues so we can fix bugs.

> **G+ Community**

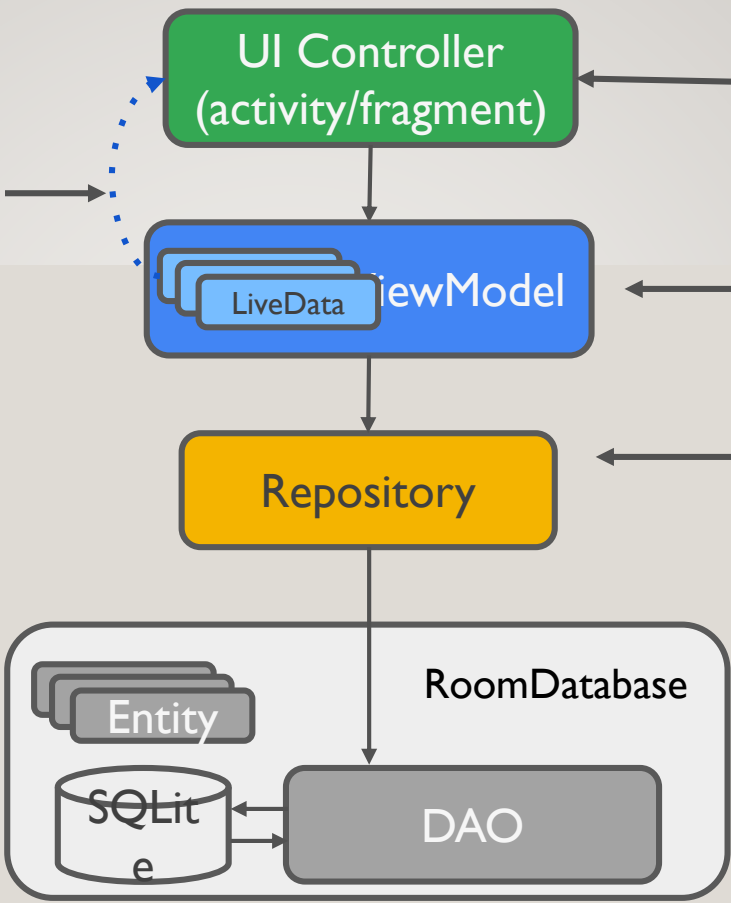
Provide feedback and discuss ideas with other developers.

In this document

- > [Common problems faced by app developers](#)
- > [Common architectural principles](#)
- > [Recommended app architecture](#)
 - > [Building the user interface](#)
 - > [Fetching data](#)
 - > [Connecting ViewModel and the repository](#)
 - > [Caching data](#)

OVERVIEW

L'UI est notifié des modifications en faisant des observations

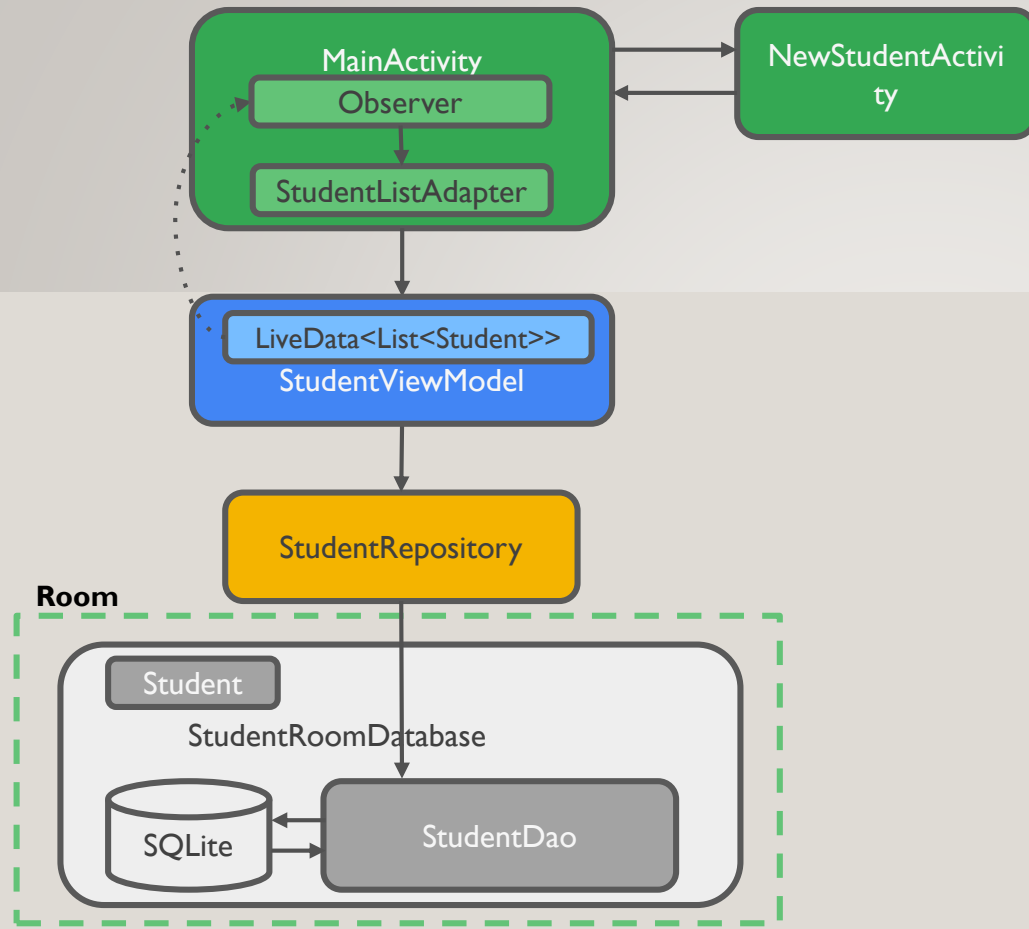


Affiche les données et gère les événements

Détient les données des UI

Une seule source d'alimentation pour les données

Gestion en local des données avec SQLite



L'application de gestion des étudiants qui vous est fournie en exemple implémente cette architecture

ROOM, LIVEDATA, VIEWMODEL

ROOM - INTRODUCTION

ROOM INTRODUCTION

ORM: Object Relational Mapper

Room est une librairie permettant de faire du mapping objet relationnel

- Génère du code Android pour manipuler SQLite
- Fournit une API simple pour la base de données



COMPOSANTS D'ARCHITECTURE DE ROOM

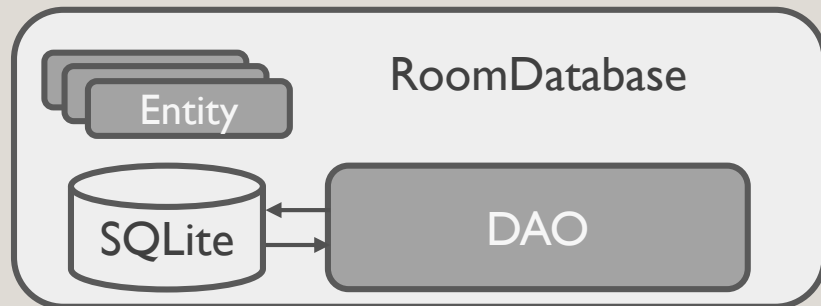
- **Entity**: Définit le schéma de la base de données.

- **DAO**: Database Access Object

Définit les opérations de lecture/écriture pour la base de données.

- **Database**:

Utilisé pour créer ou se connecter à la base de données.

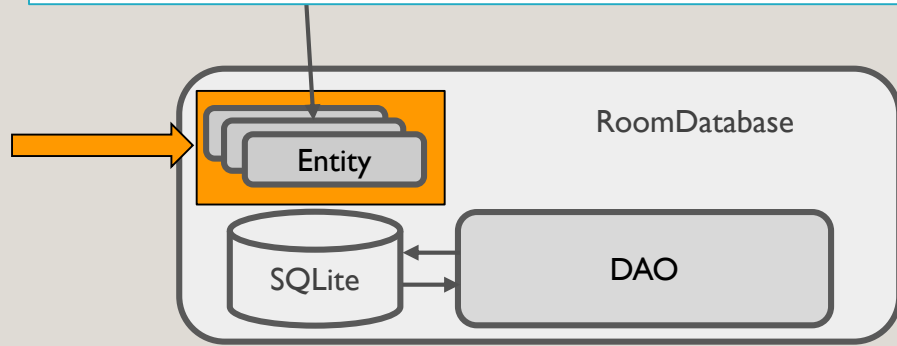


LES ENTITÉS

ENTITÉ (ENTITY)

- Entity instance = ligne dans une table de la base de données
- Définit les entités comme des classes POJO
- 1 instance = 1 ligne
- Variable membre = nom de colonne

```
class Student(  
    private val uid: Int,  
    private val firstName: String,  
    private val lastName: String  
) {  
}
```



ENTITY INSTANCE = LIGNE DANS UNE TABLE

```
class Student(  
    private val uid: Int,  
    private val firstName: String,  
    private val lastName: String  
)
```

| uid | firstName | lastName |
|-------|-----------|----------|
| 12345 | Justin | Trudeau |
| 12346 | Stephen | Harper |

ANNOTATION DES ENTITÉS

```
@Entity(tableName = "student_table")
data class Student(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name="id")
    val id:Int,
    @ColumnInfo(name = "firstName")
    val firstName: String,
    @ColumnInfo(name = "lastName")
    val lastName: String,
    @ColumnInfo(name = "phoneNumber")
    val phoneNumber: String,
    @ColumnInfo(name = "email")
    val email: String
)
```


ANNOTATION: @ENTITY

```
@Entity(tableName = "student_table")
```

- Chaque instance de l'entité représente une ligne dans une table de la base de données
- tableName spécifie un nom s'il est différent du nom de la classe.

ANNOTATION: @PRIMARYKEY

@PrimaryKey (autoGenerate=true)

- Une classe **Entity** (une **entité**) doit impérativement avoir un champ annoté comme clé primaire
- Vous pouvez générer de façon automatique une clé par entité.
- Voir [Définir des données en utilisant des entités Room](#)

ANNOTATION: @NONNULL

@NonNull

- Dénote qu'un paramètre, un champ ou une méthode ne peut pas retourner une valeur null.
- Utilisé pour des champs obligatoires
- Primary key doit utiliser @NonNull



ANNOTATION: @COLUMNINFO

```
@ColumnInfo(name = "first_name")
```

```
val firstName: String
```

```
@ColumnInfo(name = "last_name")
```

```
val lastName: String
```

- Spécifie un nom de colonne si différent du nom de la variable membre

POUR PLUS D'ANNOTATIONS

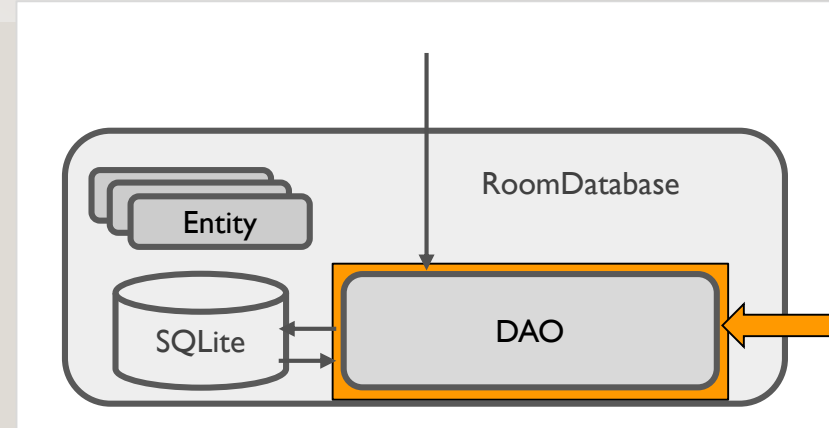
Pour plus d'annotations, voir [Room package summary reference](#)

DATA ACCESS OBJECT (DAO)

DATA ACCESS OBJECT

Utilisez les *data access objects*, ou DAOs, pour accéder aux données d'une application en utilisant librairie de persistance

Room



DATA ACCESS OBJECT

- Les méthodes DAO fournissent des accès abstraits à la base de données
- La source de données pour ces méthodes sont les entités
- DAO doit être une interface ou une classe abstraite
- Room utilise DAO pour créer une API propre pour votre code

EXAMPLE DAO

```
@Dao
interface StudentDao {
    @Query("SELECT * from student_table ORDER BY firstName ASC")
    fun getStudents(): LiveData<List<Student>>

    @Query("SELECT * FROM student_table WHERE id=(:id)")
    fun getStudent(id: Int): LiveData<Student?>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    fun insert(student: Student)

    @Update
    fun updateStudent(student: Student)

    @Query("DELETE FROM student_table")
    fun deleteAll()

    @Delete
    fun deleteStudent(student: Student)
}
```

ROOM DATABASE

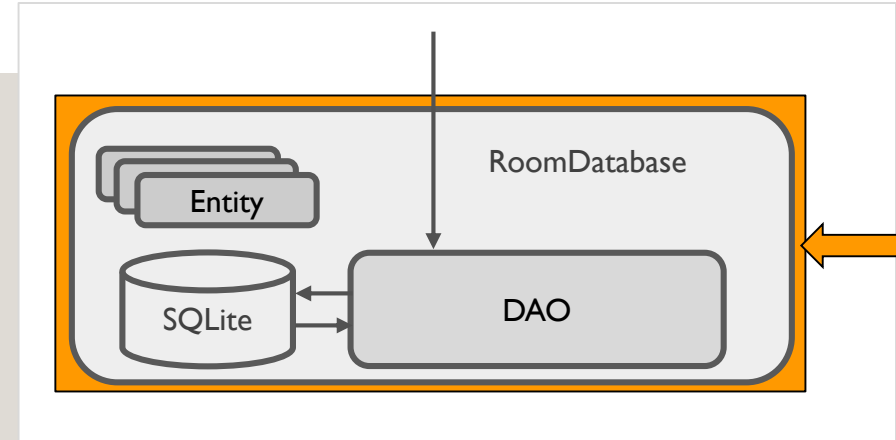
ROOM

- Room est une librairie robuste de mapping objet relationnel (ORM)
- Génère du code SQLite



ROOM

- Room travaille avec les DAO et les Entités
- Les entités définissent le schéma de la base de données
- DAO fournit les méthodes d'accès à la base de données



CRÉATION DE BASE DE DONNÉES AVEC ROOM

- Créer des classes abstraites héritant de RoomDatabase

```
abstract class StudentRoomDatabase: RoomDatabase()
```

- Annoter avec @Database
- Déclare les entités de la base de données sous forme de tableau et définit un numéro de version

```
@Database(entities = {Student.class}, version = 1)
```

```
abstract class StudentRoomDatabase: RoomDatabase()
```

EXEMPLE DE CLASSE ROOM

```
@Database(entities = {Student.class}, version =  
1)
```

```
abstract class StudentRoomDatabase  
    : RoomDatabase() {
```

*Entity définit
le schéma*

```
    abstract fun studentDao(): StudentDao;
```

*DAO pour la
BD*

```
    companion object {  
        var INSTANCE: StudentRoomDatabase? = null  
        // ... créer votre instance ici
```

*Créer la BD
comme un
singleton*

```
}
```

UTILISE UN CONSTRUCTEUR DE BD (BUILDER)

- Utilise le builder de Room pour créer la base de données
- Créer la base de données comme un singleton

```
private static StudentRoomDatabase INSTANCE;
```

```
INSTANCE = Room.databaseBuilder(...)
```

```
.build();
```

SPÉCIFIER UNE CLASSE ET UN NOM POUR LA BD

- Spécifie la classe de la base de données Room ainsi que son nom

```
INSTANCE = Room.databaseBuilder(  
    context,  
    StudentRoomDatabase.class, "student_database")  
    //...  
    .build();
```


SPÉCIFIE ONOPEN CALLBACK S'IL Y EN A

- Spécifie onOpen callback

```
INSTANCE = Room.databaseBuilder(  
    context,  
    StudentRoomDatabase.class, "student_database")  
    .addCallback(sOnOpenCallback)  
    // ...  
    .build();
```

SPÉCIFIE UNE STRATÉGIE DE MIGRATION S'IL Y EN A

- Spécifie une stratégie de migration

```
INSTANCE = Room.databaseBuilder(  
    context.getApplicationContext(),  
    StudentRoomDatabase.class, "student_database")  
    .addCallback(sOnOpenCallback)  
    .fallbackToDestructiveMigration()  
    .build();
```

EXEMPLE DE CRÉATION DE BASE DE DONNÉES ROOM

```
@Database(entities = [Student::class], version = 1, exportSchema = false)
abstract class StudentRoomDatabase: RoomDatabase() {
    abstract fun studentDao(): StudentDao

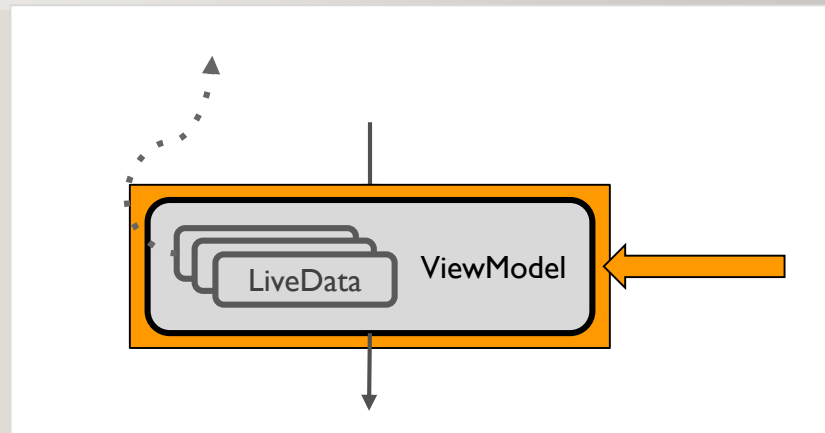
    companion object{
        @Volatile
        private var INSTANCE: StudentRoomDatabase? = null

        fun getDatabase(context: Context, scope: CoroutineScope): StudentRoomDatabase {
            val tempInstance = INSTANCE
            if(tempInstance != null)
            {
                return tempInstance
            }
            INSTANCE = Room.databaseBuilder(
                context.applicationContext,
                StudentRoomDatabase::class.java,
                "student_database"
            ).build()
            return INSTANCE as StudentRoomDatabase
        }
    }
}
```

VIEWMODEL

VIEWMODEL

- View models sont des objets qui fournissent des données pour les composantes UI et qui survivent aux changements de configuration
- Responsabilités
 - Responsable de la gestion des données pour une activité ou un fragment
 - Gère la communication entre une activité ou un fragment et le reste de l'application

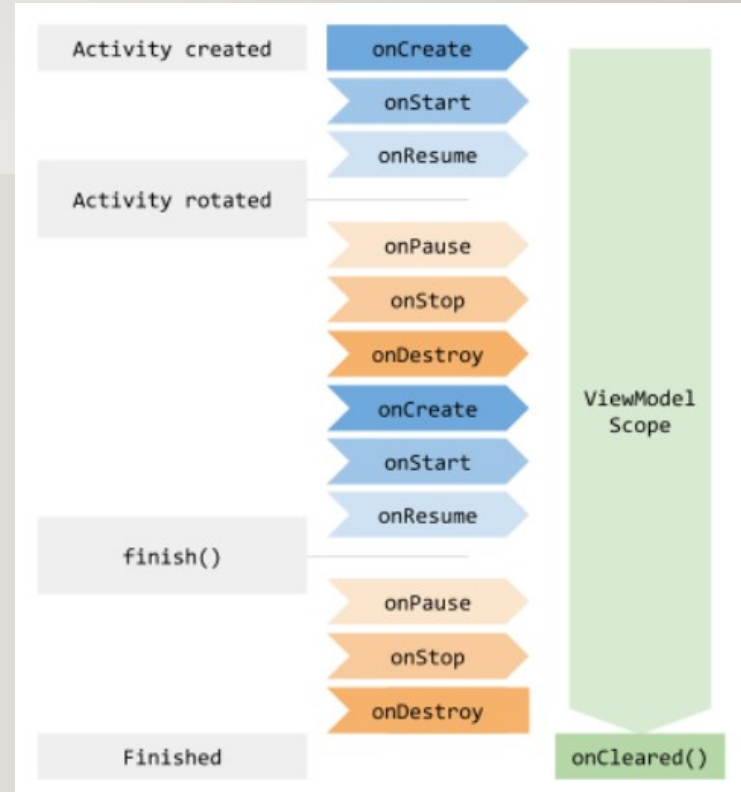


VIEWMODEL

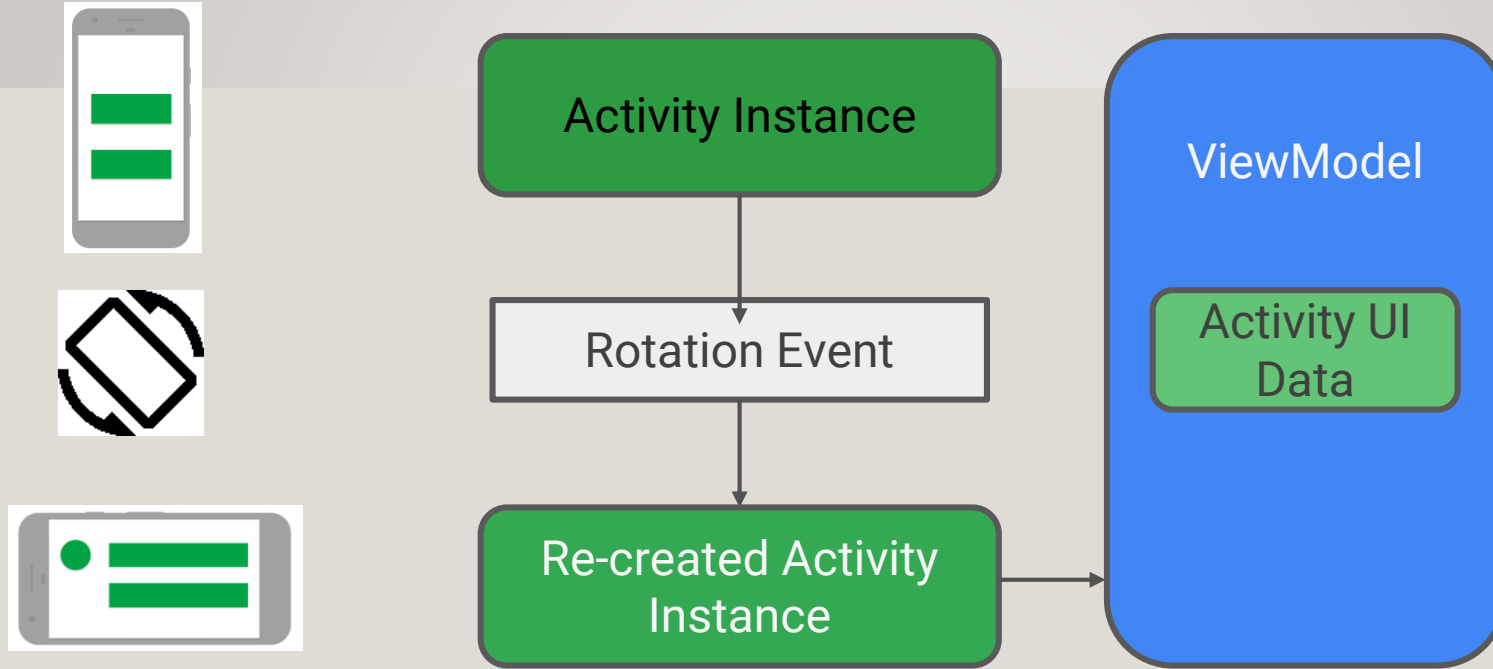
- Fournit des données à l'interface utilisateur
- Survit aux changements de configuration
- Vous pouvez aussi utiliser un ViewModel partager des données entre fragments
- Fait partie de la lifecycle library

CYCLE DE VIE D'UN VIEWMODEL

- Associé à une portée (un fragment ou une activité)
- Le ViewModel reste en vie aussi longtemps que l'activité reste en vie. Il ne sera pas détruit si l'activité ou le fragment est détruit par un changement de configuration (rotation du périphérique).
- La nouvelle instance du fragment ou de l'activité sera reconnecté au ViewModel qui existe déjà.



SURVIT AUX CHANGEMENTS DE CONFIGURATION



VIEWMODEL FOURNIT LES DONNÉES

- ViewModel fournit des données à l'interface utilisateur
- Les données peuvent provenir d'une base de données Room ou d'une autre source
- Le rôle d'un ViewModel est de retourner la donnée, il peut aider à trouver ou à générer cette donnée




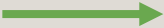
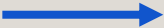

MEILLEURE PRATIQUE - UTILISER UN REPOSITORY

Recommandations de meilleures pratiques:

- Utiliser un repository pour aller chercher les données
- Garder un ViewModel comme une interface propre entre l'application et les données



ANALOGIE AVEC UN RESTAURANT

- Le client place une commande auprès du serveur 
- Le serveur passe la commande aux chefs 
- Le chef prépare le plat et le passe au serveur 
- Le serveur délivre le plat au client 
- UI demande des données au ViewModel
- ViewModel s'adresse au Repository pour les données
- Repository obtient les données et les passe au ViewModel
- ViewModel passe les données au UI

VIEWMODEL EXAMPLE

```
class StudentViewModel(application: Application): AndroidViewModel(application) {  
    private val repository StudentRepository  
    val allStudents: LiveData<List<Student>>  
    init{  
        val studentsDao = StudentRoomDatabase.getDatabase(application,  
            viewModelScope).studentDao()  
        repository = StudentRepository(studentsDao)  
        allStudents = repository.allStudents  
    }  
    fun insert(student: Student) = viewModelScope.launch(Dispatchers.IO){  
        repository.insert(student)  
    }  
}
```

VIEWMODEL EXAMPLE

```
class StudentViewModel(application: Application): AndroidViewModel(application) {  
    .. .  
    fun deleteStudent(student: Student) = viewModelScope.launch(Dispatchers.IO){  
        repository.deleteStudent(student)  
    }  
}
```

NE JAMAIS PASSEZ DE CONTEXTE AU VIEWMODEL

- Ne jamais passez de contexte (fragment ou activité) au ViewModel
- Ne pas stocker des instances d'Activité, de Fragment, ou de View ou leur Contexte dans un ViewModel
- Une activité peut être créée et détruite plusieurs fois le long du cycle de vie d'un ViewModel
- En cas de besoin d'un contexte, de [AndroidViewModel](#)

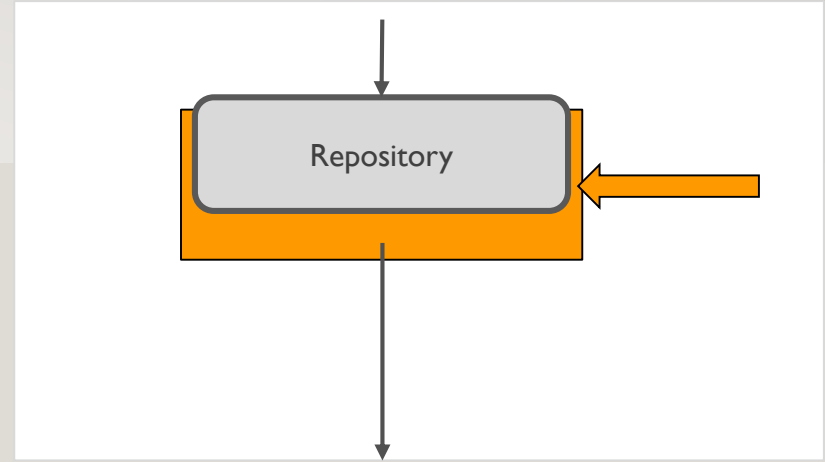
VIEWMODEL NE SURVIT PAS AU SHUTDOWN

- ViewModel survit aux modifications de configuration, mais pas au shutdown
- ViewModel ne remplace pas `onSaveInstanceState()`
(si vous ne sauvegardez pas les données avec Room)

REPOSITORY

REPOSITORY (RÉFÉRENTIEL)

- Meilleure pratique
- Ne fait pas partie des librairies de composants d'architecture (Architecture components)
- L'objectif de l'implémentation d'un repository est d'isoler la source de donnée (DAO) du ViewModel pour que le modèle ne manipule pas directement la source de donnée.

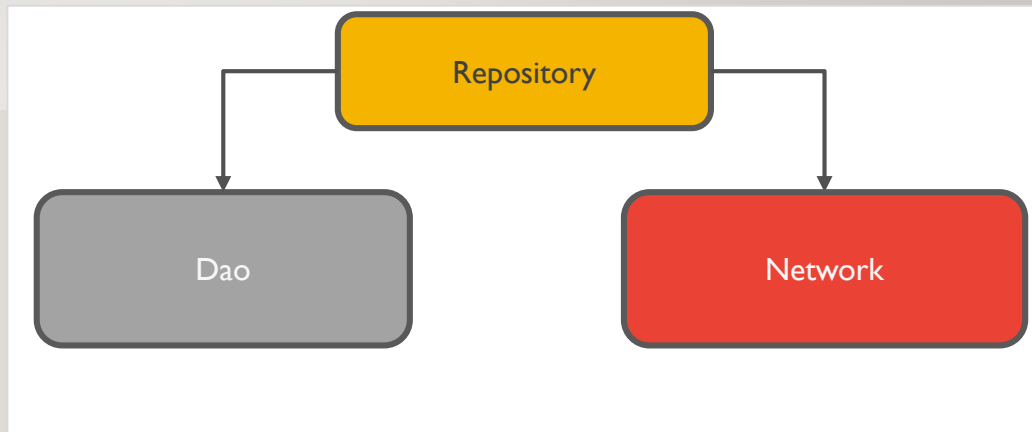


REPOSITORY CHERCHE OU GÉNÈRE DES DONNÉES

- Utiliser un repository pour chercher ou générer des données en arrière-plan
- Analogie: Les chefs qui préparent la bouffe à la cuisine

DES BACKENDS MULTIPLES

- Potentiellement, un repository vous permet de parler à différents backends comme à travers un réseau ou directement en local



EXEMPLE DE REPOSITORY

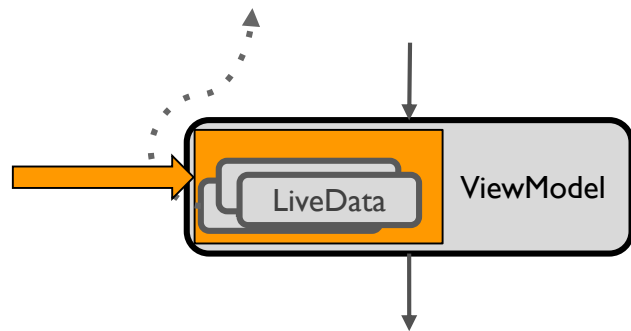
```
class StudentRepository(private val studentDao: StudentDao) {  
    // Room execute toutes les requêtes dans un thread séparé.  
    // Observed LiveData notifie l'observateur quand les données  
    changent.  
    val allStudents: LiveData<List<Student>> =  
        studentDao.getStudents()  
  
    suspend fun insert(student: Student){  
        studentDao.insert(student)  
    }  
}
```

LIVEDATA

LIVEDATA

LiveData est un data wrapper **lifecycle-aware**. C'est à dire qu'il observe l'état du composant auquel il est attaché. Ainsi, il transmettra sa donnée à son propriétaire que s'il est en mesure de le recevoir. Par exemple, une activité recevra de la data que si elle est visible et au premier plan.

Utiliser LiveData pour maintenir à jour votre interface utilisateur.



LIVEDATA

- LiveData: Données observables
- Notifie l'observateur si les données changent
- Est au courant du cycle de vie des activités/fragments Sait quand le périphérique subit une rotation
Sait quand l'application s'arrête, est en pause, etc.

UTILISER LIVEDATA POUR GARDER L'INTERFACE À JOUR

- Créer un observateur qui observe les LiveData
- LiveData notifie les objets observateurs quand les données observées changent
- L'observateur peut mettre à jour l'interface utilisateur quand les données changent.

CRÉATION DE LIVEDATA

Pour que les données soient observables, vos fonctions doivent retourner des données encapsulées par LiveData

```
@Query("SELECT * from student_table)
```

```
fun getStudents():LiveData<List<Student>>
```

UTILISER LIVEDATA AVEC ROOM

Room génère tout le code de mise à jour de LiveData quand la base de données est mise à jour

PASSER DU LIVEDATA À TRAVERS LES COUCHES

Quand vous passer du LiveData à travers les différentes couches de votre architecture, de la base de données à l'interface utilisateur, cette donnée doit être du LiveData à travers toutes les couches:

- DAO
- ViewModel
- Repository

PASSER DU LIVEDATA À TRAVERS LES COUCHES

- DAO:

```
@Query("SELECT * from word_table")  
fun getStudents():LiveData<List<Student>>
```

- Repository:

```
val allStudents: LiveData<List<Student>>  
    = studentDao.getStudents()
```

- ViewModel:

```
val allStudents: LiveData<List<Student>>  
    = repository.allStudents
```

OBSERVER DES LIVEDATA

- Créer l'observateur dans `onCreate()` dans le fragments/activité
- Modifier l'observateur pour mettre à jour l'interface utilisateur quand les données changent

Quand les données changent, l'observateur est notifié et sa méthode `onChanged()` est exécutée

OBSERVER DES LIVEDATA: EXEMPLE

```
studentViewModel.allStudents.observe(this, Observer{ students ->  
    students?.let { adapter.setStudents(it)}  
})
```

LIVEDATA EST TOUJOURS À JOUR

- Si un objet de cycle de vie devient inactif, il obtient à nouveau les données à jour quand il revient actif.
- Exemple: une activité en arrière-plan obtient ses données à jour tout de suite après qu'il revienne en avant-plan

CYCLE DEVIE

COMPOSANTS ALERTES AU CYCLE DE VIE

- Un composant avec un cycle de vie tel qu'un fragment ou une activité implémente l'interface LifecycleOwner.
- Les fragments et les activités implémentent l'interface LifecycleOwner. Ils implémentent la fonction `getLifecycle()` qui retourne le cycle de vie (objet Lifecycle) d'un fragment ou d'une activité.
- Au lieu d'avoir un fragment ou une activité gérant son cycle de vie à travers les méthodes `onStart()`, `onStop()`, etc., on peut utiliser une autre classe pour réagir au cycle de vie de ces composants.
- Les composants alertes au cycle de vie font leur travail en réponse aux modifications dans le cycle de vie d'un autre composant (fragment/activité).
- Par exemple, un listener peut être démarré ou arrêté en réponse au démarrage ou à l'arrêt d'une activité/fragment

CYCLE DE VIE (CLASSE LIFECYCLE)

- Contient des informations relatives au cycle de vie d'un composant Android
- State: Énumération représentant les différents états d'un cycle de vie d'un composant (LifecycleOwner)
 - Lifecycle.State.INITIALIZED
 - Lifecycle.State.CREATED
 - Lifecycle.State.STARTED
 - Lifecycle.State.RESUMED
 - Lifecycle.State.DESTROYED
- Events: Énumération représentant les événements du cycle de vie d'un composant (LifecycleOwner)
 - Les transitions entre les états (ON_CREATE, ON_START, ON_RESUME, ON_PAUSE, ON_STOP, ON_DESTROY, ON_ANY)

INTERFACE LIFECYCLEOBSERVER

- LifecycleObserver est une interface qui permet d'observer les composants observables, ayant un cycle de vie (LifecycleOwner) tels que les activités et les fragments.
- C'est une interface qui n'a aucune méthode. Il utilise de préférence des méthodes annotées de OnLifecycleEvent.

RÉFÉRENCES

<https://blog.ippon.fr/2018/12/01/android-architecture-components-mise-en-pratique-part-6-viewmodel-livedata/>

<https://google-developer-training.github.io/android-developer-advanced-course-practicals/unit-6-working-with-architecture-components/lesson-14-room,-livedata,-viewmodel/14-1-b-room-delete-data/14-1-b-room-delete-data.html>

FIN