Android Studio 4.0 Development Essentials

N° de la lecture individuelle :	2
Étudiant	Laurent Térence, 802_1F
Sujet	Android Studio 4.0 – Development Essentials. Smyth, Neil , Packt Publishing
Lien Livre	https://univ.scholarvox.com/catalog/book/docid/88902938?searchterm=android%20studio

Table des matières

Table des illustrations

1. Choix du sujet

J'ai décidé de me concentrer sur les bases du développement d'une application Android afin de maîtriser les fondamentaux de cette plateforme très répandue. En comprenant les concepts essentiels, les bonnes pratiques et les outils de développement, je pourrai construire des applications solides et performantes, et les améliorer au fil du temps.

En choisissant de me plonger individuellement dans ce sujet, j'ai la liberté de suivre mon propre rythme et d'approfondir les aspects qui m'intéressent le plus. Ce processus consolidera mes connaissances et me permettra d'acquérir une compréhension solide des bases du développement d'une application Android.

De plus, je suis conscient de l'importance de jeter des bases solides pour ma future carrière de développeur en me concentrant spécifiquement sur les bases. En maîtrisant les concepts essentiels, je pourrai évoluer vers des projets plus complexes et innovants, tout en évitant les erreurs courantes.

2. Aperçu de l'architecture d'Android

2.1 The Android Software Stack

Android est structuré sous la forme d'une pile logicielle comprenant des applications, un système d'exploitation, un environnement d'exécution, une infrastructure intermédiaire, des services et des bibliothèques. Cette architecture peut être représentée visuellement comme indiqué dans la *Figure 1* Chaque couche de la pile, ainsi que les éléments correspondants dans chaque couche, sont étroitement intégrées et soigneusement ajustées pour offrir un environnement optimal de développement et d'exécution des applications sur les appareils mobiles. Le reste de ce chapitre examinera les différentes couches de la pile Android, en commençant par le noyau Linux situé à la base.

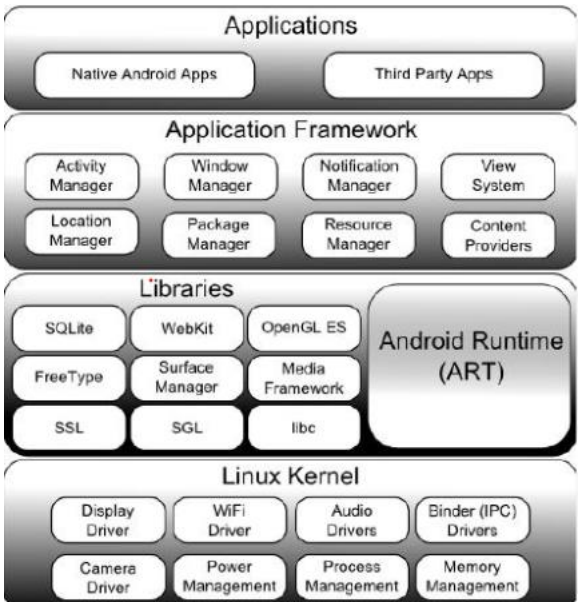

Figure 1 Android Software Stack

2.1.1 The Linux Kernel

Au bas de la pile logicielle Android se trouve le noyau Linux, qui assure une couche d'abstraction entre le matériel de l'appareil et les couches supérieures de la pile logicielle Android. Basé sur la version 2.6

de Linux, le noyau offre une gestion multitâche préemptive, des services système de base de bas niveau tels que la gestion de la mémoire, des processus et de l'alimentation, ainsi qu'une pile réseau et des pilotes de périphériques pour des composants matériels tels que l'affichage, le Wi-Fi et l'audio. Le noyau Linux original a été développé en 1991 par Linus Torvalds et a été associé à un ensemble d'outils, d'utilitaires et de compilateurs développés par Richard Stallman à la Free Software Foundation pour créer un système d'exploitation complet appelé GNU/Linux. Différentes distributions Linux ont été dérivées de ces bases fondamentales, telles qu'Ubuntu et Red Hat Enterprise Linux. Il est important de noter, cependant, qu'Android n'utilise que le noyau Linux. Cela dit, il convient de souligner que le noyau Linux a été initialement développé pour une utilisation dans des ordinateurs traditionnels tels que les ordinateurs de bureau et les serveurs. En fait, Linux est maintenant le plus largement déployé dans les environnements de serveurs d'entreprise critiques. C'est à la fois une preuve de la puissance des appareils mobiles d'aujourd'hui et de l'efficacité et des performances du noyau Linux que l'on retrouve ce logiciel au cœur de la pile logicielle Android.

2.1.2 Android Runtime – ART

Lorsqu'une application Android est construite avec Android Studio, elle est compilée dans un format intermédiaire appelé bytecode (format DEX). Lorsque l'application est ensuite chargée sur l'appareil, l'Android Runtime (ART) utilise un processus appelé compilation Ahead-of-Time (AOT) pour traduire le bytecode en instructions natives requises par le processeur de l'appareil. Ce format est connu sous le nom de format exécutable et liable (ELF). Chaque fois que l'application est lancée ultérieurement, la version exécutable ELF est utilisée, ce qui permet d'améliorer les performances de l'application et la durée de vie de la batterie. Cela contraste avec l'approche de compilation Just-in-Time (JIT) utilisée dans les anciennes versions d'Android, où le bytecode était traduit à chaque lancement de l'application dans une machine virtuelle (VM).

2.1.3 Android Librairies

En plus d'un ensemble de bibliothèques de développement Java standard (qui fournissent un support pour des tâches générales telles que la manipulation de chaînes, la gestion des réseaux et des fichiers), l'environnement de développement Android comprend également les bibliothèques Android. Ce sont un ensemble de bibliothèques basées sur Java spécifiques au développement Android. Des exemples de bibliothèques dans cette catégorie incluent les bibliothèques du framework d'application ainsi que celles qui facilitent la création d'interfaces utilisateur, le dessin graphique et l'accès à la base de données.

Voici un résumé de certaines des principales bibliothèques Android disponibles pour les développeurs Android :

- **android.app** : Permet d'accéder au modèle d'application et est la pierre angulaire de toutes les applications Android.
- **android.content** : Facilite l'accès au contenu, la publication et la communication entre les applications et les composants d'application.
- **android.database** : Utilisé pour accéder aux données publiées par les fournisseurs de contenu et inclut des classes de gestion de la base de données SQLite.
- **android.graphics** : Une API de dessin graphique 2D de bas niveau comprenant des couleurs, des points, des filtres, des rectangles et des canevas.
- **android.hardware** : Présente une API permettant d'accéder au matériel tel que l'accéléromètre et le capteur de lumière.
- **android.opengl** : Une interface Java vers l'API de rendu graphique 3D OpenGL ES.

- **android.os** : Fournit aux applications l'accès aux services standard du système d'exploitation, notamment les messages, les services système et la communication entre processus.
- **android.media** : Fournit des classes permettant la lecture de fichiers audio et vidéo.
- **android.net** : Un ensemble d'API offrant un accès à la pile réseau, incluant android.net.wi・ qui permet d'accéder à la pile sans fil de l'appareil.
- **android.print** : Inclut un ensemble de classes qui permettent l'envoi de contenu vers des imprimantes configurées à partir d'applications Android.
- **android.provider** : Un ensemble de classes pratiques qui permettent d'accéder aux bases de données standard des fournisseurs de contenu Android, telles que celles gérées par les applications de calendrier et de contacts.
- **android.text** : Utilisé pour le rendu et la manipulation du texte sur un écran de l'appareil.
- **android.util** : Un ensemble de classes utilitaires pour effectuer des tâches telles que la conversion de chaînes et de nombres, la manipulation XML et la manipulation de dates et d'heures.
- **android.view** : Les éléments fondamentaux de construction des interfaces utilisateur d'une application.
- **android.widget** : Une collection complète de composants d'interface utilisateur pré-construits tels que des boutons, des étiquettes, des listes, des gestionnaires de disposition, des boutons radio, etc.
- **android.webkit** : Un ensemble de classes destinées à permettre l'intégration de fonctions de navigation web dans les applications.

2.1.4 Application Framework

Le Framework d'application est un ensemble de services qui constituent l'environnement dans lequel les applications Android s'exécutent et sont gérées. Ce framework met en œuvre le concept selon lequel les applications Android sont construites à partir de composants réutilisables, interchangeables et remplaçables. Ce concept va encore plus loin en permettant à une application de publier ses fonctionnalités ainsi que les données correspondantes, de sorte qu'elles puissent être trouvées et réutilisées par d'autres applications. Le framework Android comprend les services clés suivants :

- Gestionnaire d'activités : Contrôle tous les aspects du cycle de vie de l'application et de la pile d'activités.
- Fournisseurs de contenu : Permet aux applications de publier et de partager des données avec d'autres applications.
- Gestionnaire de ressources : Permet d'accéder aux ressources intégrées non codées telles que les chaînes, les paramètres de couleur et les mises en page de l'interface utilisateur.
- Gestionnaire de notifications : Permet aux applications d'afficher des alertes et des notifications à l'utilisateur.
- Système de vue : Un ensemble extensible de vues utilisées pour créer des interfaces utilisateur d'application.
- Gestionnaire de packages : Le système grâce auquel les applications peuvent obtenir des informations sur les autres applications actuellement installées sur l'appareil.
- Gestionnaire de téléphonie : Fournit des informations à l'application sur les services de téléphonie disponibles sur l'appareil, tels que l'état et les informations sur l'abonné.
- Gestionnaire de localisation : Permet d'accéder aux services de localisation, ce qui permet à une application de recevoir des mises à jour sur les changements de position.

Ces services clés du framework d'Android constituent l'environnement de base dans lequel les applications Android fonctionnent et interagissent avec le système.

2.1.5 Applications

Au sommet de la pile logicielle Android se trouvent les applications. Celles-ci comprennent à la fois les applications natives fournies avec une implémentation Android spécifique (comme un navigateur web et des applications de messagerie) et les applications tierces installées par l'utilisateur après l'achat de l'appareil.

2.2 Summary

Pour développer efficacement sur Android, il est essentiel de comprendre l'architecture globale d'Android. Cette architecture repose sur une pile logicielle comprenant un noyau Linux, un environnement d'exécution et des bibliothèques correspondantes, un framework d'application et un ensemble d'applications. Les applications sont principalement écrites en Java ou en Kotlin, puis compilées en bytecode dans l'environnement de développement Android Studio. Lorsque l'application est installée sur un appareil, ce bytecode est ensuite compilé en code natif par l'Android Runtime (ART) pour être exécuté par le processeur de l'appareil. L'architecture d'Android vise principalement la performance et l'efficacité, tant dans l'exécution des applications que dans la conception permettant la réutilisation de code.

3. The Anatomy of an Android Application

L'objectif de ce chapitre est de comprendre les concepts fondamentaux de l'architecture des applications Android. Pour cela, nous explorerons en détail les différents composants utilisés pour construire une application ainsi que les mécanismes qui permettent leur interaction afin de créer une application cohérente.

4.1 Android Activities¨

Les personnes familières avec les langages de programmation orientés objet tels que Java, Kotlin, C++ ou C# seront familières avec le concept d'encapsulation des fonctionnalités d'une application dans des classes qui sont ensuite instanciées en objets et manipulées pour créer une application. Étant donné que les applications Android sont écrites en Java et Kotlin, cela reste très vrai. Cependant, Android pousse également le concept de composants réutilisables à un niveau supérieur.

Les applications Android sont créées en rassemblant un ou plusieurs composants appelés "Activities". Une activité est un module autonome de fonctionnalité d'une application qui correspond généralement directement à un écran d'interface utilisateur et à sa fonctionnalité correspondante. Par exemple, une application de rendez-vous peut avoir un écran d'activité qui affiche les rendez-vous prévus pour la journée en cours. L'application peut également utiliser une deuxième activité consistant en un écran où l'utilisateur peut saisir de nouveaux rendez-vous.

Les activités sont conçues comme des blocs de construction entièrement réutilisables et interchangeables qui peuvent être partagés entre différentes applications. Par exemple, une application de messagerie électronique existante peut contenir une activité spécifiquement conçue pour composer et envoyer un message électronique. Un développeur qui écrit une application avec une exigence similaire d'envoi de message électronique peut simplement utiliser l'activité de l'application de messagerie électronique existante, plutôt que de développer une activité spécifiquement pour la nouvelle application.

Les activités sont créées en tant que sous-classes de la classe Activity d'Android et doivent être implémentées de manière à être entièrement indépendantes des autres activités de l'application. En

d'autres termes, une activité partagée ne peut pas compter sur le fait d'être appelée à un point connu du flux du programme (puisque d'autres applications peuvent utiliser l'activité de manière imprévue) et une activité ne peut pas appeler directement les méthodes ou accéder aux données d'instance d'une autre activité. Cela est plutôt réalisé en utilisant des Intents et des Content Providers.

Par défaut, une activité ne peut pas renvoyer de résultats à l'activité à partir de laquelle elle a été appelée. Si cette fonctionnalité est requise, l'activité doit être spécifiquement lancée en tant que sous-activité de l'activité d'origine.

4.2 Android Fragments

Une activité, telle que décrite précédemment, représente généralement un écran d'interface utilisateur unique au sein d'une application. Une option consiste à construire l'activité en utilisant une seule mise en page d'interface utilisateur et un fichier de classe d'activité correspondant. Une meilleure alternative, cependant, est de diviser l'activité en différentes sections. Chacune de ces sections est appelée fragment et comprend une partie de la mise en page de l'interface utilisateur et un fichier de classe correspondant (déclaré en tant que sous-classe de la classe Fragment d'Android). Dans ce scénario, une activité devient simplement un conteneur dans lequel un ou plusieurs fragments sont intégrés.

En fait, les fragments offrent une alternative efficace à la représentation de chaque écran d'interface utilisateur par une activité distincte. Au lieu de cela, une application peut consister en une seule activité qui passe d'un fragment à un autre, chacun représentant un écran d'application différent.

4.3 Android Intents

Les Intents sont le mécanisme par lequel une activité peut en lancer une autre et mettre en œuvre le flux à travers les activités qui composent une application. Les Intents comprennent une description de l'opération à effectuer et, éventuellement, les données sur lesquelles elle doit être effectuée. Les Intents peuvent être explicites, c'est-à-dire qu'ils demandent le lancement d'une activité spécifique en référençant son nom de classe, ou implicites en précisant le type d'action à effectuer ou en fournissant des données d'un type spécifique sur lesquelles l'action doit être effectuée. Dans le cas des Intents implicites, l'exécution d'Android sélectionnera l'activité à lancer qui correspond le plus aux critères spécifiés par l'Intent en utilisant un processus appelé Résolution d'Intent.

Les Intents explicites sont utilisés lorsque vous souhaitez lancer une activité spécifique de manière directe et précise, en référençant son nom de classe. Cela offre un contrôle plus granulaire sur le flux de l'application.

D'un autre côté, les Intents implicites sont utiles lorsque vous voulez déléguer la responsabilité de choisir l'activité appropriée au système Android. Vous spécifiez simplement l'action à effectuer ou les données sur lesquelles l'action doit être réalisée, et le système sélectionnera l'activité la plus adaptée en fonction des filtres d'intent définis par les applications installées sur le dispositif. Cela peut être pratique lorsque vous souhaitez offrir à l'utilisateur la possibilité de choisir parmi plusieurs applications pouvant effectuer une action donnée.

En résumé, les Intents explicites offrent un contrôle direct tandis que les Intents implicites offrent une plus grande flexibilité et une meilleure intégration avec d'autres applications. Le choix dépendra donc des fonctionnalités et de l'expérience utilisateur que vous souhaitez offrir dans votre application.

4.4 Broadcast Intents

Les Broadcast Intents sont un autre type d'Intent utilisé pour envoyer des messages à toutes les applications qui ont enregistré un "Broadcast Receiver" intéressé. Le système Android, par exemple,

envoie souvent des Broadcast Intents pour indiquer des changements d'état du dispositif, tels que la fin du démarrage du système, la connexion d'une source d'alimentation externe ou l'allumage/éteignage de l'écran.

Un Broadcast Intent peut être normal (asynchrone), c'est-à-dire qu'il est envoyé à tous les Broadcast Receivers intéressés en même temps, ou ordonné, ce qui signifie qu'il est envoyé à un récepteur à la fois, où il peut être traité puis soit interrompu soit transmis au récepteur suivant.

En résumé, les Broadcast Intents permettent d'envoyer des messages à plusieurs applications en même temps, ce qui les rend utiles pour la diffusion d'informations système ou pour la coordination de diverses actions entre applications.

4.5 Broadcast Receivers

Les Broadcast Receivers sont le mécanisme par lequel les applications peuvent répondre aux Broadcast Intents. Un Broadcast Receiver doit être enregistré par une application et configuré avec un filtre d'intent pour indiquer les types de diffusion qui l'intéressent. Lorsqu'un intent correspondant est diffusé, le récepteur est invoqué par le système Android, que l'application qui a enregistré le récepteur soit en cours d'exécution ou non. Le récepteur dispose ensuite de 5 secondes pour effectuer les tâches qui lui sont assignées (telles que le lancement d'un Service, la mise à jour des données ou l'émission d'une notification à l'utilisateur) avant de retourner. Les Broadcast Receivers fonctionnent en arrière-plan et n'ont pas d'interface utilisateur.

En résumé, les Broadcast Receivers permettent aux applications de réagir aux Broadcast Intents enregistrés. Ils peuvent effectuer des actions en réponse à des événements système, même si l'application n'est pas en cours d'exécution à ce moment-là. Cela les rend utiles pour effectuer des tâches en arrière-plan telles que la synchronisation de données, le traitement des notifications ou le déclenchement d'autres services.

4.6 Android Services

Les services Android sont des processus qui s'exécutent en arrière-plan et n'ont pas d'interface utilisateur. Ils peuvent être démarrés et gérés ultérieurement à partir d'activités, de Broadcast Receivers ou d'autres services. Les services Android sont idéaux dans les situations où une application doit continuer à effectuer des tâches sans nécessiter nécessairement une interface utilisateur visible. Bien que les services ne disposent pas d'une interface utilisateur, ils peuvent tout de même informer l'utilisateur d'événements à l'aide de notifications et de toasts (de petits messages de notification qui apparaissent à l'écran sans interrompre l'activité actuellement visible) et sont également en mesure d'émettre des Intents.

Les services ont une priorité plus élevée que de nombreux autres processus pour le système Android et ne seront terminés par celui-ci qu'en dernier recours afin de libérer des ressources. Dans le cas où le système doit effectivement arrêter un service, il sera automatiquement redémarré dès que des ressources suffisantes seront à nouveau disponibles. Un service peut réduire le risque de terminaison en se déclarant comme devant s'exécuter en premier plan. Cela est réalisé en appelant startForeground(). Cependant, cela est recommandé uniquement dans les situations où la terminaison serait préjudiciable à l'expérience utilisateur (par exemple, si l'utilisateur écoute de l'audio diffusé par le service).

Des exemples de situations où un service peut être une solution pratique incluent, comme mentionné précédemment, la diffusion en continu d'audio qui doit se poursuivre lorsque l'application n'est plus

active, ou une application de suivi de la bourse qui doit avertir l'utilisateur lorsqu'une action atteint un prix spécifié.

En résumé, les services Android sont des processus en arrière-plan qui permettent à une application de continuer à effectuer des tâches sans interface utilisateur visible. Ils peuvent émettre des notifications, des toasts et des Intents, et ont une priorité élevée dans le système. Ils sont utiles pour des fonctionnalités telles que la diffusion en continu d'audio ou les notifications basées sur des événements en arrière-plan.

4.7 Content Providers

Les Content Providers sont des mécanismes permettant le partage de données entre les applications. Une application peut fournir aux autres applications un accès à ses données sous-jacentes en mettant en place un Content Provider, y compris la possibilité d'ajouter, de supprimer et de consulter les données (sous réserve des autorisations). L'accès aux données se fait via un identifiant de ressource universel (URI) défini par le Content Provider. Les données peuvent être partagées sous la forme d'un fichier ou d'une base de données SQLite complète.

Les applications natives Android incluent plusieurs Content Providers standard qui permettent aux applications d'accéder aux données telles que les contacts et les fichiers multimédias. Les Content Providers disponibles sur un système Android peuvent être localisés à l'aide d'un Content Resolver.

En résumé, les Content Providers sont des mécanismes permettant de partager des données entre applications. Ils offrent la possibilité d'ajouter, de supprimer et de consulter des données, et peuvent partager des fichiers ou des bases de données SQLite. Les applications natives Android proposent des Content Providers standard pour accéder à des données telles que les contacts et les médias. Le Content Resolver permet de localiser les Content Providers disponibles sur le système Android.

4.8 The Application Manifest

Le fichier Manifeste de l'application est la pièce maîtresse qui rassemble les différents éléments constitutifs d'une application. C'est dans ce fichier, basé sur XML, que l'application décrit les activités, les services, les récepteurs de diffusion, les fournisseurs de données et les autorisations qui composent l'application complète.

En résumé, le fichier Manifeste de l'application est essentiel pour rassembler les différents éléments d'une application. Il permet de définir les activités, les services, les récepteurs de diffusion, les fournisseurs de données et les autorisations nécessaires à l'application.

4.9 Application Ressources

En plus du fichier manifeste et des fichiers **Dex** contenant le bytecode, un package d'application Android contient généralement une collection de fichiers de ressources. Ces fichiers contiennent des ressources telles que les chaînes de caractères, les images, les polices et les couleurs qui apparaissent dans l'interface utilisateur, ainsi que la représentation XML des mises en page de l'interface utilisateur. Par défaut, ces fichiers sont stockés dans le sous-répertoire **/res** de la hiérarchie du projet d'application.

En résumé, un package d'application Android comprend également des fichiers de ressources qui contiennent les éléments visuels et les mises en page de l'interface utilisateur de l'application, tels que les chaînes de caractères, les images et les couleurs. Ces fichiers sont stockés dans le répertoire /res du projet d'application.

4.10 Application Context

Lorsqu'une application est compilée, une classe nommée R est créée et contient des références aux ressources de l'application. Le fichier manifeste de l'application et ces ressources se combinent pour créer ce qu'on appelle le Contexte de l'Application. Ce contexte, représenté par la classe Android Context, peut être utilisé dans le code de l'application pour accéder aux ressources de l'application lors de l'exécution. De plus, une grande variété de méthodes peut être appelées sur le contexte de l'application pour recueillir des informations et apporter des modifications à l'environnement de l'application lors de l'exécution.

En résumé, lors de la compilation d'une application, une classe appelée R est créée pour faire référence aux ressources de l'application. Le fichier manifeste de l'application et ces ressources se combinent pour former le Contexte de l'Application, qui permet d'accéder aux ressources et de modifier l'environnement de l'application lors de l'exécution. Le contexte de l'application offre également de nombreuses méthodes pour obtenir des informations sur l'application.

4.11 Summary

Ce chapitre nous a donné un aperçu global des différents éléments qui permettent de créer une application Android. Nous avons abordé les activités (Activities), les fragments, les services (Services), les intents et les récepteurs de diffusion (Broadcast Receivers), ainsi que le fichier manifeste et les ressources de l'application.

L'objectif est de favoriser la réutilisation maximale et l'interopérabilité en créant des modules de fonctionnalité individuels et autonomes sous forme d'activités et d'intents, tandis que le partage de données entre les applications est réalisé grâce à la mise en œuvre de fournisseurs de contenu (content providers). Les activités se concentrent sur les interactions de l'utilisateur avec l'application, chaque activité correspondant généralement à une seule interface utilisateur, composée éventuellement de fragments. Les traitements en arrière-plan sont généralement gérés par les services et les récepteurs de diffusion.

Les composants de l'application sont définis dans un fichier manifeste, qui, combiné aux ressources de l'application, représente le contexte de l'application. Bien que ces concepts puissent être nouveaux pour de nombreux développeurs, nous explorerons et utiliserons concrètement ces concepts dans les chapitres suivants pour vous permettre de construire vos propres applications sur une base solide de connaissances.

5 An Overview of Android View Binding

L'interaction entre le code et les vues qui composent les mises en page de l'interface utilisateur est une partie essentielle du développement d'applications Android. Ce chapitre se concentrera sur les différentes options disponibles pour accéder aux vues de la mise en page dans le code, en mettant particulièrement l'accent sur une option introduite avec Android Studio 3.6 appelée "view binding" (liaison de vues). Une fois les bases de la liaison de vues expliquées, le chapitre présentera les modifications nécessaires pour convertir le projet AndroidSample afin d'utiliser cette approche.

5.1 Find View by ID

Comme expliqué dans le chapitre intitulé "La structure d'une application Android", toutes les ressources qui composent une application sont compilées dans une classe appelée R. Parmi ces ressources se trouvent celles qui définissent les mises en page. Dans la classe R, il existe une sous-classe nommée "layout" qui contient les ressources de mise en page, y compris les vues qui composent l'interface utilisateur. La plupart des applications auront besoin d'interagir entre le code et ces vues,

par exemple lors de la lecture de la valeur saisie dans une vue EditText ou lors de la modification du contenu affiché dans un TextView.

Avant l'introduction d'Android Studio 3.6, la seule option pour accéder à une vue à partir du code de l'application consistait à écrire du code pour rechercher manuellement une vue en fonction de son identifiant via une méthode appelée "findViewById()". Par exemple :

TextView exampleView = findViewById(R.id.exampleView);

Une fois la référence obtenue, les propriétés de la vue peuvent être utilisées. Par exemple :

exampleView.setText("Bonjour");

Bien que la recherche de vues par identifiant soit toujours une option viable, elle présente certaines limites. Le plus grand inconvénient de "findViewById()" est qu'il est possible d'obtenir une référence à une vue qui n'a pas encore été créée dans la mise en page, ce qui entraîne une exception de pointeur null lorsqu'une tentative est faite pour accéder aux propriétés de la vue. Depuis Android Studio 3.6, une autre méthode d'accès aux vues depuis le code de l'application est disponible sous la forme de "view bindings" (liaisons de vues).

5.2 View Bindings

Lorsque les liaisons de vues sont activées dans un module d'application, Android Studio génère automatiquement une classe de liaison pour chaque fichier de mise en page du module. En utilisant cette classe de liaison, les vues de la mise en page peuvent être accessibles depuis le code sans avoir besoin d'utiliser "findViewById()". Le nom de la classe de liaison générée par Android Studio est basé sur le nom du fichier de mise en page converti en "camel case" avec le mot "Binding" ajouté à la fin.

Par exemple, dans le cas du fichier activity_main.xml, la classe de liaison sera nommée ActivityMainBinding.

Le processus d'utilisation des liaisons de vues dans un module de projet peut être résumé comme suit :

1. Activer les liaisons de vues pour les modules de projet nécessitant une prise en charge.

2. Modifier le code pour importer la classe de liaison de vue auto-générée.

3. Instancier la classe de liaison pour obtenir une référence à la liaison.

4. Accéder à la vue racine à l'intérieur de la liaison et l'utiliser pour spécifier la vue de contenu de l'interface utilisateur.

5. Accéder aux vues par leur nom en tant que propriétés de l'objet de liaison.

5.3 Enabling View Binding

Rajouter dans le fichier Build.gardle(Module:app):

```
android{
        buildFeatures{
                viewBinding = true
        }
}
```

5.4 Using View Bindings

La première étape de ce processus consiste à "inflater" la classe de liaison de vue afin d'accéder à la vue racine de la mise en page. Cette vue racine sera ensuite utilisée comme vue de contenu pour la mise en page. L'endroit logique pour effectuer ces tâches est à l'intérieur de la méthode onCreate() de l'activité associée à la mise en page. Une méthode onCreate() typique se présente comme suit :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
}
```

Pour passer à l'utilisation des liaisons de vues, la classe de liaison de vue doit être importée et la classe modifiée comme suit. Notez que puisque le fichier de mise en page est nommé activity_main.xml, nous pouvons supposer que la classe de liaison générée par Android Studio sera nommée **ActivityMainBinding :**

```
import android.widget.EditText;
import android.widget.TextView;
import com.example.androidsample.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {
   private ActivityMainBinding binding;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      binding = ActivityMainBinding.inflate(getLayoutInflater());
      View view = binding.getRoot();
      setContentView(view);
   }
}
```

Maintenant que nous avons une référence à la liaison de vue, nous pouvons accéder aux vues par leur nom de la manière suivante :

```
public void convertCurrency(View view) {
```

```
    EditText dollarText = binding.dollarText;
    TextView textView = binding.textView;


    if (!dollarText.getText().toString().equals("")) {
        Float dollarValue = Float.valueOf(dollarText.getText().toString());
        Float euroValue = dollarValue * 0.85F;
        textView.setText(euroValue.toString());
    } else {
        textView.setText(R.string.no_value_string);
    }
}
```

5.5 Choosing an Option¨

L'introduction des liaisons de vues ne rend pas obsolètes les options précédentes, et les deux méthodes continueront d'être largement prises en charge dans un avenir prévisible. En termes d'évitement des exceptions de pointeur nul, cependant, les liaisons de vues sont clairement l'option la plus sûre par rapport à l'utilisation de la méthode `findViewById()`. Lorsque vous développez vos propres projets, il est donc recommandé d'utiliser les liaisons de vues.

Cependant, il est important de noter que les liaisons de vues ne sont pas activées par défaut dans Android Studio 4.0 pour les exemples de ce livre. Malheureusement, pour utiliser les liaisons de vues dans les exemples du livre, il serait nécessaire de répéter manuellement chacune des modifications du fichier `build.gradle` et de la méthode `onCreate()` de ce chapitre plus de 50 fois. Pour cette raison, les exemples de ce livre continuent d'utiliser `findViewById()`.

5.6 Summary

Avant l'introduction d'Android Studio 3.6, pour accéder aux vues de mise en page depuis le code d'une application, on utilisait la méthode findViewById(). Une alternative est maintenant disponible sous la forme de view bindings. Les view bindings sont des classes automatiquement générées par Android Studio pour chaque fichier de mise en page XML. Ces classes contiennent des liaisons vers chacune des vues de la mise en page correspondante, offrant ainsi une option plus sûre que la méthode findViewById(). Cependant, depuis Android Studio 3.6, les view bindings ne sont pas activés par défaut et des étapes supplémentaires sont nécessaires pour activer et configurer manuellement leur prise en charge dans chaque module de projet.

6 Understanding Android Application and Activity Lifecycles

6.1 Introduction

Ce chapitre vise à approfondir nos connaissances sur le cycle de vie des applications et des activités dans le système d'exécution Android. Nous avons déjà appris dans les chapitres précédents que les applications Android s'exécutent dans des processus et sont composées de plusieurs éléments tels que les activités, les services et les récepteurs de diffusion. Malgré les performances accrues des appareils mobiles d'aujourd'hui par rapport aux systèmes de bureau d'hier, il est important de garder à l'esprit que ces appareils restent considérés comme étant "contraints par les ressources" par rapport aux

ordinateurs de bureau modernes, notamment en termes de mémoire. Ainsi, un des rôles clés du système Android est de gérer efficacement ces ressources limitées et de garantir la réactivité du système d'exploitation et des applications à tout moment. Pour y parvenir, Android a un contrôle total sur le cycle de vie et l'état des processus dans lesquels s'exécutent les applications, ainsi que sur les composants individuels de ces applications. Il est donc essentiel de comprendre les modèles de gestion du cycle de vie des applications et des activités Android, ainsi que les moyens pour une application de réagir aux changements d'état qui peuvent survenir tout au long de son exécution.

6.2 Android Applications and Resource Management

Chaque application Android en cours d'exécution est considérée par le système d'exploitation comme un processus distinct. Lorsque le système détecte que les ressources de l'appareil atteignent leur capacité maximale, il prend des mesures pour terminer certains processus afin de libérer de la mémoire. Lorsqu'il décide quel processus terminer pour libérer de la mémoire, le système tient compte à la fois de la priorité et de l'état de tous les processus en cours d'exécution, et combine ces facteurs pour créer ce que Google appelle une hiérarchie d'importance. Les processus sont alors terminés en commençant par ceux ayant la plus basse priorité, jusqu'à ce que suffisamment de ressources aient été libérées pour assurer le bon fonctionnement du système.

6.3 Android Process States

Les processus hébergent des applications et les applications sont constituées de composants. Dans un système Android, l'état actuel d'un processus est défini par le composant actif de plus haut rang au sein de l'application qu'il héberge. Comme indiqué dans la Figure 2, un processus peut se trouver dans l'un des cinq états suivants à tout moment.

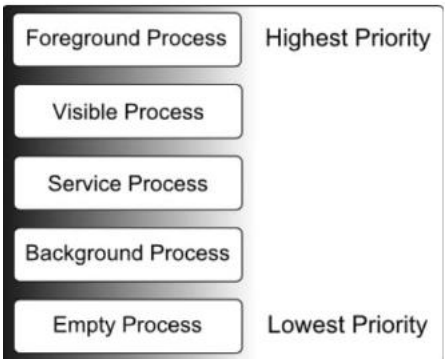

Figure 2 : Android Process Priority

6.3.1 Foreground Process

Ces processus sont attribués au plus haut niveau de priorité. À un moment donné, il y a rarement plus d'un ou deux processus en premier plan actifs et ce sont généralement les derniers à être terminés par le système. Un processus doit remplir un ou plusieurs des critères suivants pour être considéré comme étant en premier plan :

- Héberge une activité avec laquelle l'utilisateur interagit actuellement.
- Héberge un service connecté à l'activité avec laquelle l'utilisateur interagit.
- Héberge un service qui a indiqué, via un appel à startForeground(), que sa terminaison serait perturbatrice pour l'expérience utilisateur.
- Héberge un service exécutant ses méthodes onCreate(), onResume() ou onStart().
- Héberge un récepteur de diffusion qui exécute actuellement sa méthode onReceive().

6.3.2 Visible Process

Un processus contenant une activité qui est visible pour l'utilisateur mais avec laquelle l'utilisateur n'interagit pas est classé comme un "processus visible". C'est généralement le cas lorsqu'une activité du processus est visible pour l'utilisateur, mais qu'une autre activité, telle qu'un écran partiel ou une boîte de dialogue, est en premier plan. Un processus est également éligible au statut visible s'il héberge un service qui est lui-même lié à une activité visible ou en premier plan.

6.3.3 Service Process

Les processus qui contiennent un service qui a déjà été démarré et qui s'exécute actuellement.

6.3.4 Background Process

Un processus qui contient une ou plusieurs activités qui ne sont pas visibles pour l'utilisateur et n'héberge pas de service qualifiant pour le statut de processus de service. Les processus de cette catégorie sont à haut risque de terminaison en cas de besoin de libérer de la mémoire pour des processus de priorité supérieure. Android maintient une liste dynamique de processus en arrière-plan, en terminant les processus dans l'ordre chronologique, de sorte que les processus qui étaient les moins récemment utilisés sont tués en premier.

6.3.5 Empty Process

Les processus vides ne contiennent plus d'applications actives et sont conservés en mémoire pour servir de plateformes d'accueil pour de nouvelles applications lancées. Cela ressemble un peu à maintenir les portes ouvertes et le moteur en marche dans un bus en prévision de l'arrivée des passagers. Ces processus sont évidemment considérés comme étant de la plus basse priorité et sont les premiers à être arrêtés pour libérer des ressources.

6.4 Inter-Process Dependencies

La situation concernant la détermination du processus de priorité la plus élevée est légèrement plus complexe que ce qui a été décrit dans la section précédente, pour la simple raison que les processus peuvent souvent être interdépendants. Ainsi, lorsqu'il s'agit de déterminer la priorité d'un processus, le système Android prend également en compte le fait que ce processus sert d'une manière ou d'une autre un autre processus de priorité supérieure (par exemple, un processus de service agissant en tant que fournisseur de contenu pour un processus en premier plan). En règle générale, la documentation Android stipule qu'un processus ne peut jamais être classé plus bas qu'un autre processus qu'il sert actuellement.

6.5 The Activity Lifecycle

Comme nous l'avons déjà déterminé, l'état d'un processus Android est largement déterminé par le statut des activités et des composants qui constituent l'application qu'il héberge. Il est donc important de comprendre que ces activités passent également par différents états au cours de l'exécution de l'application. L'état actuel d'une activité est déterminé, en partie, par sa position dans ce qu'on appelle la pile d'activités.

6.6 The Activity Stack

Chaque application en cours d'exécution sur un appareil Android possède une pile d'activités, gérée par le système d'exécution. Lorsqu'une application est lancée, sa première activité est placée au sommet de la pile. Lorsqu'une deuxième activité est démarrée, elle est placée au-dessus de la pile et pousse l'activité précédente vers le bas. L'activité en tête de pile est l'activité active (ou en cours d'exécution). Lorsque cette activité active se termine, elle est retirée de la pile par le système

d'exécution et l'activité immédiatement en dessous devient l'activité active. L'activité en tête de pile peut sortir simplement parce que la tâche pour laquelle elle est responsable est terminée. De plus, l'utilisateur peut appuyer sur le bouton "Retour" pour revenir à l'activité précédente, ce qui entraîne la suppression de l'activité active de la pile par le système d'exécution et donc sa destruction. La pile d'activités Android est représentée visuellement dans la Figure 3. Les nouvelles activités sont ajoutées au sommet de la pile lorsqu'elles sont démarrées. L'activité active se trouve au sommet de la pile jusqu'à ce qu'une nouvelle activité la pousse vers le bas ou qu'elle soit retirée de la pile lorsqu'elle se termine ou lorsque l'utilisateur revient à l'activité précédente. En cas de limitation des ressources, le système d'exécution supprimera des activités en commençant par celles situées en bas de la pile. La pile d'activités est une structure de données de type "Last-In-First-Out" (LIFO) : le dernier élément ajouté est le premier à être retiré.

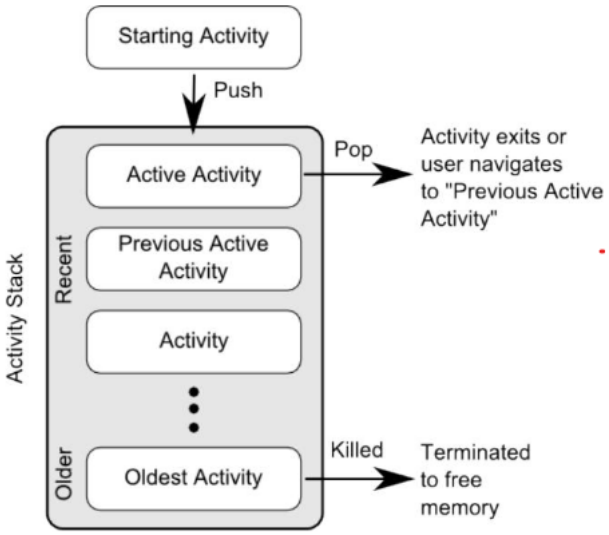

Figure 3 Android Activity Stack

6.7 Activity States

Une activité peut se trouver dans différents états au cours de son exécution dans une application :

- Active / En cours d'exécution : l'activité est en tête de la pile d'activités, elle est la tâche en cours affichée à l'écran, a le focus et interagit actuellement avec l'utilisateur. C'est l'activité la moins susceptible d'être terminée en cas de manque de ressources.
- En pause : l'activité est visible pour l'utilisateur mais n'a pas actuellement le focus (généralement parce qu'elle est partiellement cachée par l'activité active en cours). Les activités en pause sont conservées en mémoire, restent attachées au gestionnaire de fenêtres, conservent toutes les informations d'état et peuvent rapidement être restaurées en tant qu'activité active lorsqu'elles sont placées en tête de la pile d'activités.
- Arrêtée : l'activité n'est pas actuellement visible pour l'utilisateur (elle est totalement masquée sur l'écran par d'autres activités). Comme les activités en pause, elle conserve toutes les informations d'état et de membre, mais présente un risque plus élevé d'être terminée en cas de faible mémoire.
- Terminée : l'activité a été arrêtée par le système d'exécution pour libérer de la mémoire et n'est plus présente dans la pile d'activités. Ces activités doivent être redémarrées si nécessaire par l'application.

6.8 Configuration Changes

Dans ce chapitre, nous avons examiné deux des causes de changement d'état d'une activité Android, à savoir le passage d'une activité entre l'avant-plan et l'arrière-plan, et la terminaison d'une activité par le système d'exécution pour libérer de la mémoire. En réalité, il existe un troisième scénario dans lequel l'état d'une activité peut changer de manière significative, et cela implique un changement de configuration de l'appareil.

Par défaut, tout changement de configuration qui affecte l'apparence d'une activité (comme la rotation de l'orientation de l'appareil entre le mode portrait et paysage, ou la modification des paramètres de police du système) entraînera la destruction et la recréation de l'activité. La raison derrière cela est que de tels changements affectent des ressources telles que la disposition de l'interface utilisateur, et détruire et recréer simplement les activités impactées est le moyen le plus rapide pour qu'une activité réagisse au changement de configuration. Cependant, il est possible de configurer une activité de manière qu'elle ne soit pas redémarrée par le système en réponse à des changements de configuration spécifiques.

6.9 Handling State Changes

Ce chapitre met en évidence le fait qu'une application, et par définition les composants qu'elle contient, traversent de nombreux états tout au long de leur cycle de vie. Il est important de noter que ces changements d'état, allant jusqu'à la terminaison complète, sont imposés à l'application par le système d'exécution Android en fonction des actions de l'utilisateur et de la disponibilité des ressources sur l'appareil.

Cependant, ces changements d'état ne sont généralement pas imposés sans préavis. En pratique, le système d'exécution notifie souvent l'application des changements et lui donne l'opportunité de réagir en conséquence. Cela implique généralement la sauvegarde ou la restauration des structures de données internes et de l'état de l'interface utilisateur, permettant ainsi à l'utilisateur de passer facilement d'une application à une autre et donnant l'apparence d'applications fonctionnant simultanément.

Android propose deux méthodes pour gérer les changements d'état des objets au sein d'une application. L'une consiste à répondre aux appels de méthodes de changement d'état du système d'exploitation, et elle est détaillée dans le prochain chapitre intitulé "Gestion des changements d'état d'activité Android". Une nouvelle approche, recommandée par Google, utilise les classes de cycle de vie incluses dans les composants d'architecture Jetpack pour Android, introduits dans "Architecture moderne des applications Android avec Jetpack" et expliqués plus en détail dans le chapitre intitulé "Travailler avec les composants Android Lifecycle-Aware".

6.10 Summary

Les appareils mobiles sont généralement considérés comme ayant des ressources limitées, notamment en termes de capacité de mémoire interne. Par conséquent, l'une des principales responsabilités du système d'exploitation Android est de veiller à ce que les applications, ainsi que le système d'exploitation lui-même, restent réactifs à l'utilisateur. Les applications sont hébergées sur Android au sein de processus. Chaque application est composée de différentes activités et services. Le système d'exécution d'Android a le pouvoir de terminer à la fois des processus et des activités individuelles afin de libérer de la mémoire. L'état d'un processus est pris en compte par le système d'exécution lorsqu'il décide si un processus peut être terminé. L'état d'un processus dépend largement de l'état des activités hébergées par ce processus. Le message clé de ce chapitre est que chaque application passe par différents états tout au long de son cycle de vie d'exécution et a peu de contrôle sur son destin au sein de l'environnement d'exécution Android. Les processus et activités qui

n'interagissent pas directement avec l'utilisateur courent un plus grand risque d'être terminés par le système d'exécution. Il est donc essentiel, dans le développement d'applications Android, que l'application puisse réagir aux notifications de changement d'état émises par le système d'exploitation.

7 Handling Android Activity State Changes

7.1 The Android Lifecycle Methods

Les classes Activity et Fragment d'Android disposent de plusieurs méthodes de cycle de vie qui agissent comme des gestionnaires d'événements lorsque l'état d'une instance change. Les principales méthodes prises en charge par les classes Activity et Fragment d'Android sont les suivantes :

- onCreate(Bundle savedInstanceState): Appelée lors de la création initiale de l'activité, cette méthode est l'endroit idéal pour effectuer la plupart des tâches d'initialisation. Un objet Bundle peut être passé en argument et contenir des informations d'état dynamique de l'interface utilisateur d'une invocation précédente de l'activité.
- onRestart(): Appelée lorsque l'activité est sur le point de redémarrer après avoir été précédemment arrêtée par le système d'exécution.
- onStart(): Toujours appelée immédiatement après les méthodes onCreate() ou onRestart(), cette méthode indique à l'activité qu'elle est sur le point de devenir visible pour l'utilisateur. Elle sera suivie d'un appel à onResume() si l'activité passe en premier plan, ou onStop() si elle est repoussée vers l'arrière-plan par une autre activité.
- onResume(): Indique que l'activité est maintenant en haut de la pile des activités et que c'est l'activité avec laquelle l'utilisateur interagit actuellement.
- onPause(): Indique qu'une activité précédente est sur le point de devenir l'activité en premier plan. Cet appel sera suivi soit par onResume() soit par onStop(), selon que l'activité revient à l'avant-plan ou devient invisible pour l'utilisateur. Des mesures peuvent être prises dans cette méthode pour stocker des informations d'état persistantes non encore enregistrées par l'application.
- onStop(): L'activité n'est plus visible pour l'utilisateur. Deux scénarios possibles peuvent suivre cet appel : un appel à onRestart() si l'activité passe à nouveau en premier plan, ou onDestroy() si l'activité est en cours de terminaison.
- onDestroy(): L'activité est sur le point d'être détruite, soit volontairement parce qu'elle a terminé ses tâches et a appelé la méthode finish(), soit parce que le système d'exécution la termine pour libérer de la mémoire ou en raison d'un changement de configuration.
- onConfigurationChanged(): Appelée lorsqu'un changement de configuration se produit pour lequel l'activité a indiqué qu'elle ne devait pas être redémarrée. La méthode reçoit un objet Configuration décrivant la nouvelle configuration de l'appareil, et il revient à l'activité de réagir à ce changement.

En plus de ces méthodes de cycle de vie, il existe deux méthodes spécifiquement destinées à sauvegarder et restaurer l'état dynamique d'une activité :

- onRestoreInstanceState(Bundle savedInstanceState): Cette méthode est appelée immédiatement après un appel à onStart() si l'activité redémarre à partir d'une invocation précédente où l'état a été enregistré. Comme pour onCreate(), cette méthode reçoit un objet Bundle contenant les données d'état précédentes.
- onSaveInstanceState(Bundle outState): Appelée avant la destruction d'une activité pour sauvegarder l'état dynamique actuel (généralement lié à l'interface utilisateur). La méthode

reçoit l'objet Bundle dans lequel l'état doit être sauvegardé et qui est ensuite transmis aux méthodes onCreate() et onRestoreInstanceState() lorsque l'activité est redémarrée. Il est important de noter que cette méthode n'est appelée que lorsque le système d'exécution estime que l'état dynamique doit être sauvegardé.

Lors de la substitution des méthodes mentionnées ci-dessus, il est important de se rappeler que, à l'exception de onRestoreInstanceState() et onSaveInstanceState(), l'implémentation de la méthode doit inclure un appel à la méthode correspondante de la superclasse. Par exemple, la méthode onRestart() peut être ainsi redéfinie :

```
protected void onRestart() {
   super.onRestart();
   Log.i(TAG, "onRestart");
}
```

L'absence de cet appel à la superclasse dans les substitutions de méthode entraînera une exception lors de l'exécution. Bien que les appels à la superclasse dans les méthodes onRestoreInstanceState() et onSaveInstanceState() soient facultatifs (ils peuvent être omis, par exemple, lors de la mise en œuvre d'un comportement de sauvegarde et de restauration personnalisé), il est avantageux de les utiliser, un sujet qui sera abordé dans le chapitre intitulé "Sauvegarde et restauration de l'état d'une activité Android".

7.2 Lifetimes

Le dernier sujet à aborder concerne un aperçu des durées de vie complète, visible et au premier plan à travers lesquelles une activité ou un fragment va passer pendant l'exécution :

- Durée de vie complète : C'est la période qui s'étend entre l'appel initial à la méthode onCreate() et l'appel à onDestroy() avant que l'objet ne se termine.
- Durée de vie visible : Couvre les périodes d'exécution entre les appels à onStart() et onStop(). Pendant cette période, l'activité ou le fragment est visible pour l'utilisateur, même s'il ne s'agit pas forcément de l'objet avec lequel l'utilisateur interagit actuellement.
- Durée de vie au premier plan : Fait référence aux périodes d'exécution entre les appels aux méthodes onResume() et onPause().

Il est important de noter qu'une activité ou un fragment peut passer plusieurs fois par les durées de vie au premier plan et visible au cours de sa durée de vie complète. Les concepts de durées de vie et de méthodes du cycle de vie sont illustrés dans la Figure 4.

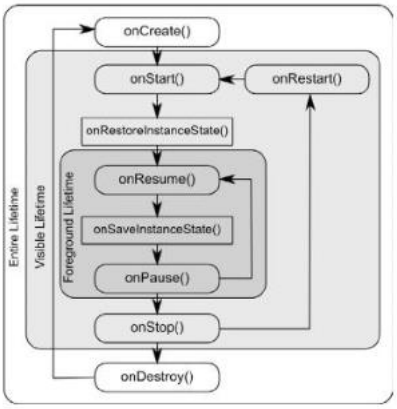

Figure 4 Lifetimes and lifecycle methods

7.3 Lifecycle Method Limitations

Comme discuté au début de ce chapitre, les méthodes du cycle de vie sont utilisées depuis de nombreuses années et, jusqu'à récemment, constituaient le seul mécanisme disponible pour gérer les changements d'état du cycle de vie des activités et des fragments. Cependant, cette approche présente certaines limites.

Une des limitations des méthodes du cycle de vie est qu'elles ne permettent pas à une activité ou à un fragment de connaître facilement son état actuel à un moment donné de l'exécution de l'application. L'objet devrait donc suivre son état en interne ou attendre l'appel de la prochaine méthode du cycle de vie. De plus, ces méthodes ne fournissent pas un moyen simple pour qu'un objet puisse observer les changements d'état du cycle de vie d'autres objets au sein de l'application. Cela est particulièrement important, car de nombreux autres objets au sein de l'application peuvent potentiellement être impactés par un changement d'état du cycle de vie dans une activité ou un fragment donné.

Les méthodes du cycle de vie sont également disponibles uniquement pour les sous-classes des classes Fragment et Activity. Il n'est donc pas possible de créer des classes personnalisées qui sont réellement conscientes du cycle de vie. Enfin, les méthodes du cycle de vie entraînent la majorité du code de gestion du cycle de vie écrit dans l'activité ou le fragment lui-même, ce qui peut conduire à un code complexe et sujet aux erreurs. Idéalement, une grande partie de ce code devrait résider dans les autres classes impactées par le changement d'état.

Tous ces problèmes et bien d'autres sont résolus en utilisant des composants conscients du cycle de vie, un sujet qui sera abordé à partir du chapitre intitulé "Modern Android App Architecture with Jetpack".

7.4 Summary

Toutes les activités sont dérivées de la classe Activity d'Android, qui contient à son tour plusieurs méthodes du cycle de vie conçues pour être appelées par le système d'exécution lorsque l'état d'une activité change. De même, la classe Fragment contient plusieurs méthodes comparables. En les substituant, les activités et les fragments peuvent réagir aux changements d'état et, si nécessaire, prendre des mesures pour sauvegarder et restaurer l'état actuel de l'activité et de l'application. L'état du cycle de vie peut être divisé en deux formes : l'état persistant, qui fait référence aux données devant être conservées entre les invocations de l'application (par exemple, dans un fichier ou une base de données), et l'état dynamique, qui concerne l'apparence actuelle de l'interface utilisateur.

Bien que les méthodes du cycle de vie présentent certaines limites pouvant être évitées en utilisant des composants conscients du cycle de vie, il est important de comprendre ces méthodes pour appréhender pleinement les nouvelles approches de gestion du cycle de vie abordées ultérieurement dans ce livre.

8 An Introduction to Android Fragments

8.1 What is a Fragment ?

Un fragment est une section autonome et modulaire de l'interface utilisateur d'une application, ainsi que du comportement qui lui est associé, pouvant être intégré dans une activité. Les fragments peuvent être assemblés pour créer une activité lors de la conception de l'application, et ajoutés ou supprimés d'une activité pendant l'exécution de l'application pour créer une interface utilisateur qui change dynamiquement.

Les fragments ne peuvent être utilisés que dans le cadre d'une activité et ne peuvent pas être instanciés en tant qu'éléments d'application autonomes. Cela étant dit, un fragment peut être considéré comme un "sous-activité" fonctionnel avec son propre cycle de vie similaire à celui d'une activité complète.

Les fragments sont stockés sous la forme de fichiers de mise en page XML et peuvent être ajoutés à une activité soit en plaçant les éléments <fragment> appropriés dans le fichier de mise en page de l'activité, soit directement à travers le code de mise en œuvre de la classe de l'activité.

8.2 Creating a Fragment

Les fragments sont composés de deux éléments : un fichier de mise en page XML et une classe Java correspondante. Le fichier de mise en page XML d'un fragment a le même format qu'une mise en page pour n'importe quelle autre activité et peut contenir n'importe quelle combinaison et complexité de gestionnaires de mise en page et de vues. Par exemple, le fichier de mise en page XML suivant est pour un fragment composé simplement d'un RelativeLayout avec un arrière-plan rouge contenant un seul TextView :

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/red" >

  <TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
```

```
    android:layout_centerVertical="true"
    android:text="@string/fragone_label_text"
    android:textAppearance="?android:attr/textAppearanceLarge" />
</RelativeLayout>
```

La classe correspondante à la mise en page doit être une sous-classe de la classe Fragment d'Android. Cette classe doit au moins remplacer la méthode onCreateView(), qui est responsable du chargement de la mise en page du fragment. Par exemple :

```java
package com.example.myfragmentdemo;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;

public class FragmentOne extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    // Inflater la mise en page pour ce fragment
    return inflater.inflate(R.layout.fragment_one_layout, container, false);
  }
}
```

En plus de la méthode onCreateView(), la classe peut également remplacer les méthodes du cycle de vie standard. Une fois que la mise en page et la classe du fragment ont été créées, le fragment est prêt à être utilisé dans les activités de l'application.

8.3 Adding a Frament to an Activity using the Layout XML File

Un fragment peut être incorporé dans une activité soit en écrivant du code Java, soit en intégrant le fragment dans le fichier de mise en page XML de l'activité. Peu importe l'approche utilisée, il est important de noter que lorsqu'une bibliothèque de compatibilité est utilisée pour prendre en charge les anciennes versions d'Android, toutes les activités utilisant des fragments doivent être

implémentées en tant que sous-classe de FragmentActivity plutôt que de la classe AppCompatActivity :

```java
package com.example.myfragmentdemo;

import android.os.Bundle;
import androidx.fragment.app.FragmentActivity;
import android.view.Menu;

public class MainActivity extends FragmentActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_fragment_demo);
   }
}
```

Les fragments sont intégrés dans les fichiers de mise en page des activités à l'aide de l'élément `<fragment>`. L'exemple de mise en page suivant intègre le fragment créé dans la section précédente de ce chapitre dans une mise en page d'activité :

```xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

   <fragment
      android:id="@+id/fragment_one"
      android:name="com.example.myfragmentdemo.myfragmentdemo.FragmentOne"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
```

android:layout_alignParentLeft="true"

android:layout_centerVertical="true"

tools:layout="@layout/fragment_one_layout" />

</RelativeLayout>

```

Les propriétés clés de l'élément `<fragment>` sont `android:name`, qui doit faire référence à la classe associée au fragment, et `tools:layout`, qui doit faire référence au fichier de ressources XML contenant la mise en page du fragment. Une fois ajoutés à la mise en page d'une activité, les fragments peuvent être visualisés et manipulés dans l'outil Android Studio Layout Editor.

## 8.4 Adding and Managing Fragments in Code

L'ajout d'un fragment à une activité via le fichier de mise en page XML de l'activité facilite son intégration, mais empêche la suppression du fragment pendant l'exécution. Pour avoir un contrôle dynamique complet des fragments pendant l'exécution, il est nécessaire d'ajouter les fragments via du code. Cela permet d'ajouter, de supprimer et même de remplacer dynamiquement les fragments pendant que l'application est en cours d'exécution.

Lorsqu'on utilise du code pour gérer les fragments, le fragment lui-même est toujours composé d'un fichier de mise en page XML et d'une classe correspondante. La différence réside dans la façon de travailler avec le fragment dans l'activité hôte. Il existe une séquence d'étapes standard pour ajouter un fragment à une activité en utilisant du code :

**1. Créer une instance de la classe du fragment.**

**2. Transmettre d'éventuels arguments supplémentaires via l'instance de classe.**

**3. Obtenir une référence à l'instance du gestionnaire de fragments.**

**4. Appeler la méthode beginTransaction() sur l'instance du gestionnaire de fragments. Cela renvoie une instance de transaction de fragment.**

**5. Appeler la méthode add() de l'instance de transaction de fragment, en passant en argument l'ID de la vue qui doit contenir le fragment et l'instance de classe du fragment.**

**6. Appeler la méthode commit() de la transaction de fragment.**

Par exemple, le code suivant ajoute un fragment défini par la classe FragmentOne pour qu'il apparaisse dans la vue de conteneur avec l'ID LinearLayout1 :

**```java**

**FragmentOne firstFragment = new FragmentOne();**

**firstFragment.setArguments(getIntent().getExtras());**

**FragmentManager fragManager = getSupportFragmentManager();**

**FragmentTransaction transaction = fragManager.beginTransaction();**
```

transaction.add(R.id.LinearLayout1, firstFragment);

transaction.commit();

```

Le code ci-dessus décompose chaque étape en une instruction distincte pour des raisons de clarté. Cependant, les quatre dernières lignes peuvent être abrégées en une seule ligne de code comme suit :

```**java**

getSupportFragmentManager().beginTransaction()

.add(R.id.LinearLayout1, firstFragment).commit();

```

Une fois ajouté à un conteneur, un fragment peut être ultérieurement supprimé en appelant la méthode remove() de l'instance de transaction de fragment, en passant en argument une référence à l'instance du fragment à supprimer :

```**java**

transaction.remove(firstFragment);

```

De même, un fragment peut être remplacé par un autre en appelant la méthode replace() de l'instance de transaction de fragment. Cela prend en argument l'ID de la vue contenant le fragment et une instance du nouveau fragment. Le fragment remplacé peut également être placé dans ce qu'on appelle la pile arrière (back stack) afin de pouvoir être rapidement restauré si l'utilisateur y revient. Cela s'obtient en appelant la méthode addToBackStack() de l'objet de transaction de fragment avant d'appeler la méthode commit() :

```**java**

FragmentTwo secondFragment = new FragmentTwo();

transaction.replace(R.id.LinearLayout1, secondFragment);

transaction.addToBackStack(null);

transaction.commit();

```

## 8.5 Handling Fragment Events

Comme mentionné précédemment, un fragment est similaire à une sous-activité avec sa propre mise en page, sa classe et son cycle de vie. Les composants de vue (comme les boutons et les textes) à l'intérieur d'un fragment peuvent générer des événements de la même manière que dans une activité classique. Cela soulève la question de savoir quelle classe reçoit un événement d'une vue dans un fragment : le fragment lui-même ou l'activité dans laquelle le fragment est intégré. La réponse à cette question dépend de la façon dont le gestionnaire d'événements est déclaré.
```

Dans le chapitre intitulé "Aperçu et exemple de la gestion des événements Android", deux approches de gestion des événements ont été discutées. La première méthode consiste à configurer un écouteur d'événements et une méthode de rappel dans le code de l'activité. Par exemple :

```java
button.setOnClickListener(
    new Button.OnClickListener() {
        public void onClick(View v) {
            // Code à exécuter lorsque le bouton est cliqué
        }
    }
);
```

Dans le cas de l'interception des événements de clic, la deuxième approche consiste à définir la propriété android:onClick dans le fichier de mise en page XML :

```xml
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onClick"
    android:text="Cliquez ici"
/>
```

La règle générale pour les événements générés par une vue dans un fragment est que si l'écouteur d'événements a été déclaré dans la classe du fragment en utilisant l'approche de l'écouteur d'événements et de la méthode de rappel, alors l'événement sera d'abord traité par le fragment. En revanche, si la ressource **android:onClick** est utilisée, l'événement sera directement transmis à l'activité contenant le fragment.

8.6 Implementing Fragment Communication

Une fois qu'un ou plusieurs fragments sont intégrés dans une activité, il est fort probable qu'une forme de communication soit nécessaire à la fois entre les fragments et l'activité, et entre un fragment et un autre. En fait, il est recommandé de ne pas permettre aux fragments de communiquer directement entre eux. Toute communication doit passer par l'activité qui les encapsule. Pour qu'une activité communique avec un fragment, elle doit identifier l'objet fragment via l'ID qui lui est attribué. Une fois cette référence obtenue, l'activité peut simplement appeler les méthodes publiques de l'objet fragment.

La communication dans l'autre sens (du fragment vers l'activité) est un peu plus compliquée. Dans un premier temps, le fragment doit définir une interface d'écouteur, qui est ensuite implémentée dans la classe de l'activité. Par exemple, le code suivant déclare une interface nommée ToolbarListener dans une classe de fragment nommée ToolbarFragment. Le code déclare également une variable dans laquelle une référence à l'activité sera stockée ultérieurement :

```java
public class ToolbarFragment extends Fragment {
   ToolbarListener activityCallback;

   public interface ToolbarListener {
      public void onButtonClick(int position, String text);
   }
   ...
}
```

Le code ci-dessus indique que toute classe qui implémente l'interface ToolbarListener doit également implémenter une méthode de rappel nommée onButtonClick, qui accepte à son tour un entier et une chaîne de caractères en tant qu'arguments. Ensuite, la méthode onAttach() de la classe de fragment doit être substituée et implémentée. Cette méthode est appelée automatiquement par le système Android lorsque le fragment a été initialisé et associé à une activité. La méthode reçoit une référence à l'activité dans laquelle le fragment est contenu. La méthode doit stocker une référence locale à cette activité et vérifier qu'elle implémente l'interface ToolbarListener :

```java
@Override
public void onAttach(Context context) {
   super.onAttach(context);
   try {
      activityCallback = (ToolbarListener) activity;
   } catch (ClassCastException e) {
      throw new ClassCastException(activity.toString()
            + " must implement ToolbarListener");
   }
}
```

Lors de l'exécution de cet exemple, une référence à l'activité sera stockée dans la variable locale activityCallback, et une exception sera levée si cette activité n'implémente pas l'interface ToolbarListener.

La prochaine étape consiste à appeler la méthode de rappel de l'activité depuis le fragment. Quand et comment cela se produit dépend entièrement des circonstances dans lesquelles l'activité doit être contactée par le fragment. Par exemple, le code suivant appelle la méthode de rappel sur l'activité lorsqu'un bouton est cliqué :

```java
public void buttonClicked (View view) {
   activityCallback.onButtonClick(arg1, arg2);
}
```

Il ne reste plus qu'à modifier la classe de l'activité pour qu'elle implémente l'interface ToolbarListener. Par exemple :

```java
public class MainActivity extends FragmentActivity implements ToolbarFragment.ToolbarListener {
   public void onButtonClick(String arg1, int arg2) {
      // Implémenter le code de la méthode de rappel
   }
   ...
}
```

Comme nous pouvons le voir dans le code ci-dessus, l'activité déclare qu'elle implémente l'interface ToolbarListener de la classe ToolbarFragment, puis elle procède à l'implémentation de la méthode **onButtonClick()** comme requis par l'interface.

Ainsi, en suivant cette approche, les fragments peuvent communiquer avec l'activité en appelant les méthodes publiques de l'objet fragment, tandis que l'activité peut communiquer avec les fragments en implémentant l'interface définie par le fragment et en fournissant une implémentation appropriée des méthodes de rappel.

Cela garantit une communication efficace et structurée entre les fragments et l'activité, tout en respectant les bonnes pratiques de développement Android.

8.7 Summary

Les fragments offrent un mécanisme puissant pour créer des modules réutilisables de mise en page de l'interface utilisateur et de comportement d'application, qui, une fois créés, peuvent être intégrés dans des activités. Un fragment est composé d'un fichier de mise en page de l'interface utilisateur et d'une classe. Les fragments peuvent être utilisés dans une activité en ajoutant le fragment au fichier de mise en page de l'activité, ou en écrivant du code pour gérer les fragments dynamiquement à l'exécution.

Les fragments ajoutés à une activité en code peuvent être supprimés et remplacés dynamiquement à l'exécution. Toute communication entre les fragments doit être effectuée via l'activité dans laquelle les fragments sont intégrés.

Ayant couvert les bases des fragments dans ce chapitre, le chapitre suivant présentera un tutoriel conçu pour renforcer les techniques décrites dans ce chapitre.

9 Modern Android App Architecture with Jetpack

9.1 What is Android Jetpack?

Android Jetpack est composé d'Android Studio, des composants d'architecture Android et de la bibliothèque de support Android, ainsi que d'un ensemble de lignes directrices qui recommandent comment une application Android devrait être structurée. Les composants d'architecture Android sont conçus pour permettre une réalisation plus rapide et plus facile des tâches courantes lors du développement d'applications Android, tout en respectant le principe clé des lignes directrices architecturales.

Bien que tous les composants d'architecture Android soient abordés dans ce livre, l'objectif de ce chapitre est de présenter les principales lignes directrices architecturales, ainsi que les composants ViewModel, LiveData, Lifecycle, l'utilisation de Data Binding et des référentiels. Avant de passer à la suite, il est important de comprendre que l'approche Jetpack pour le développement d'applications n'est pas obligatoire. Tout en soulignant les limites d'autres techniques qui ont gagné en popularité au fil des ans, Google n'a pas complètement condamné ces approches de développement d'applications. Google semble adopter la position selon laquelle il n'y a pas de bonne ou de mauvaise façon de développer une application, mais une façon recommandée.

9.2 Modern Android Architecture

À un niveau très basique, Google préconise désormais des applications à activité unique où différents écrans sont chargés en tant que contenu au sein de la même activité. Les lignes directrices modernes en matière d'architecture recommandent également de séparer les différentes responsabilités au sein d'une application en modules entièrement distincts (un concept que Google appelle "séparation des préoccupations"). L'un des éléments clés de cette approche est le composant ViewModel.

9.3 The ViewModel Component

Le ViewModel a pour objectif de séparer le modèle de données et la logique liée à l'interface utilisateur de l'application du code chargé d'afficher et de gérer réellement l'interface utilisateur et d'interagir avec le système d'exploitation. Lorsqu'il est conçu de cette manière, une application se compose d'un ou plusieurs contrôleurs d'interface utilisateur, tels qu'une activité, associés à des instances de ViewModel chargées de gérer les données nécessaires à ces contrôleurs. En effet, le ViewModel ne connaît que le modèle de données et la logique correspondante. Il ne sait rien de l'interface utilisateur et ne tente pas d'accéder directement aux événements relatifs aux vues de l'interface utilisateur. Lorsqu'un contrôleur d'interface utilisateur a besoin de données à afficher, il demande simplement au ViewModel de les fournir. De même, lorsque l'utilisateur saisit des données dans une vue de l'interface utilisateur, le contrôleur d'interface utilisateur les transmet au ViewModel pour les traiter.

Cette séparation des responsabilités permet de résoudre les problèmes liés au cycle de vie des contrôleurs d'interface utilisateur. Peu importe combien de fois un contrôleur d'interface utilisateur est recréé pendant le cycle de vie d'une application, les instances de ViewModel restent en mémoire, garantissant ainsi la cohérence des données. Par exemple, un ViewModel utilisé par une activité

restera en mémoire jusqu'à ce que l'activité se termine complètement, ce qui, dans une application à activité unique, n'arrive pas avant la sortie de l'application.

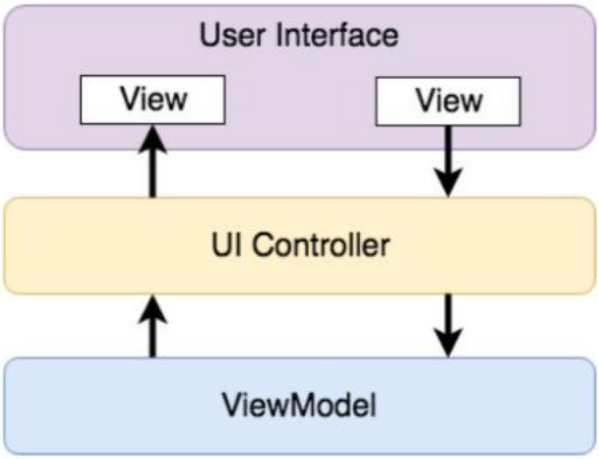

Figure 5 ViewModel Component

9.4 The LiveData Component

Imaginez une application qui affiche en temps réel des données telles que le prix actuel d'une action financière. Cette application utiliserait probablement un service web de prix des actions pour mettre à jour en continu le modèle de données à l'intérieur du ViewModel avec les dernières informations. Évidemment, ces données en temps réel sont peu utiles si elles ne sont pas affichées à l'utilisateur de manière rapide. Il existe seulement deux façons pour le contrôleur de l'interface utilisateur de garantir que les dernières données sont affichées dans l'interface utilisateur. Une option consiste pour le contrôleur à vérifier en continu auprès du ViewModel si les données ont changé depuis leur dernier affichage. Cependant, cette approche est inefficace. Pour maintenir la nature en temps réel des données, le contrôleur de l'interface utilisateur devrait fonctionner en boucle, vérifiant en permanence les changements des données.

Une meilleure solution serait pour le contrôleur de l'interface utilisateur de recevoir une notification lorsqu'un élément de données spécifique dans un ViewModel change. Cela est rendu possible grâce à l'utilisation du composant LiveData. LiveData est un conteneur de données qui permet à une valeur d'être observée. En termes simples, un objet observable a la capacité de notifier d'autres objets lorsque des changements surviennent dans ses données, résolvant ainsi le problème d'assurer que l'interface utilisateur correspond toujours aux données du ViewModel.

Cela signifie, par exemple, qu'un contrôleur de l'interface utilisateur qui est intéressé par une valeur du ViewModel peut mettre en place un observateur qui sera notifié lorsque cette valeur change. Dans notre application hypothétique, par exemple, le prix de l'action serait enveloppé dans un objet LiveData à l'intérieur du ViewModel, et le contrôleur de l'interface utilisateur attribuerait un observateur à cette valeur, déclarant une méthode à appeler lorsque la valeur change. Cette méthode, lorsqu'elle est déclenchée par un changement de données, lirait la valeur mise à jour depuis le ViewModel et l'utiliserait pour mettre à jour l'interface utilisateur.

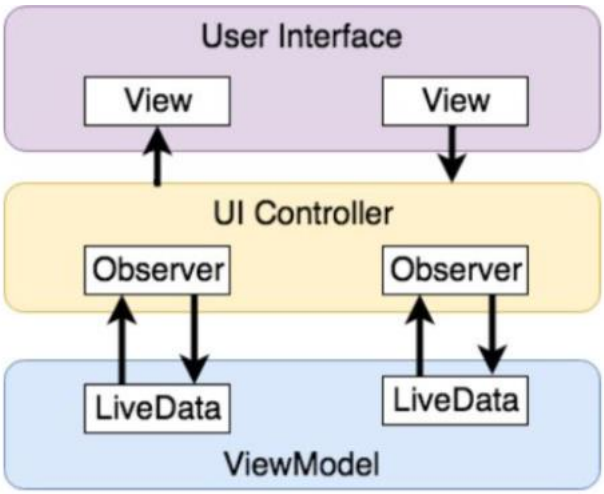

Figure 6 LiveData Component

Une instance de LiveData peut également être déclarée comme étant mutable, permettant à l'entité observatrice de mettre à jour la valeur sous-jacente contenue dans l'objet LiveData. Par exemple, l'utilisateur peut entrer une valeur dans l'interface utilisateur qui doit écraser la valeur stockée dans le ViewModel.

Un autre avantage clé de l'utilisation de LiveData est qu'il est conscient de l'état du cycle de vie de ses observateurs. Par exemple, si une activité contient un observateur LiveData, l'objet LiveData correspondant saura quand l'état du cycle de vie de l'activité change et réagira en conséquence. Si l'activité est en pause (par exemple, si l'application passe en arrière-plan), l'objet LiveData cessera d'envoyer des événements à l'observateur. Si l'activité vient de démarrer ou reprend après une pause, l'objet LiveData enverra un événement LiveData à l'observateur pour que l'activité dispose de la valeur la plus récente. De même, l'instance LiveData saura quand l'activité est détruite et supprimera l'observateur pour libérer des ressources.

Jusqu'à présent, nous avons seulement parlé des contrôleurs d'interface utilisateur utilisant des observateurs. En pratique, cependant, un observateur peut être utilisé dans n'importe quel objet qui se conforme à l'approche de Jetpack en matière de gestion du cycle de vie.

9.5 ViewModel Saved State

Sur Android, l'utilisateur a la possibilité de mettre une application active en arrière-plan et d'y revenir plus tard après avoir effectué d'autres tâches sur l'appareil (y compris l'exécution d'autres applications). Lorsqu'un appareil manque de ressources, le système d'exploitation remédie à cela en terminant les processus des applications en arrière-plan, en commençant par celle utilisée le moins récemment. Cependant, lorsque l'utilisateur revient à une application en arrière-plan qui a été terminée, elle devrait apparaître dans le même état que lorsqu'elle a été mise en arrière-plan, indépendamment de sa terminaison. En ce qui concerne les données associées à un ViewModel, cela peut être mis en œuvre en utilisant le module ViewModel Saved State. Ce module permet de stocker des valeurs dans l'état enregistré de l'application et de les restaurer en cas de terminaison initiée par le système, un sujet qui sera abordé ultérieurement dans le chapitre intitulé "Un tutoriel sur l'état enregistré d'un ViewModel Android".

9.6 LiveData and Data Binding

Android Jetpack comprend la bibliothèque de liaison de données qui permet de mapper directement les données d'un ViewModel à des vues spécifiques dans le fichier de mise en page XML de l'interface utilisateur. Dans le projet AndroidSample créé précédemment, du code devait être écrit à la fois pour obtenir les références des vues EditText et TextView, et pour définir et obtenir les propriétés de texte afin de refléter les changements de données. La liaison de données permet de référencer directement la valeur LiveData stockée dans le ViewModel dans le fichier de mise en page XML, évitant ainsi la nécessité d'écrire du code pour maintenir les vues de mise en page à jour.

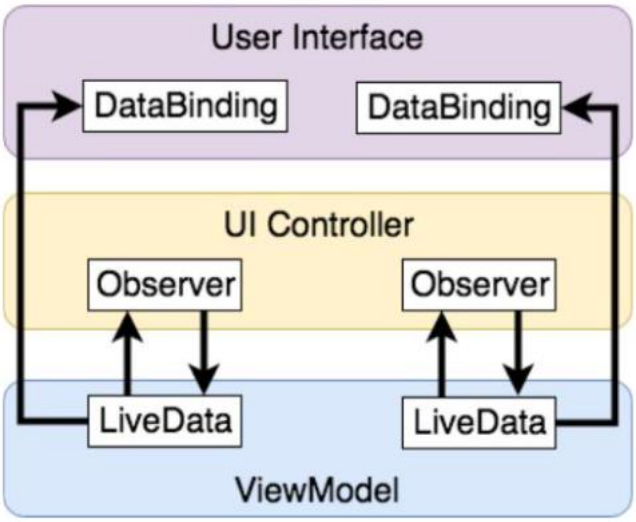

Figure 7 LiveData & DataBinding

9.7 Repository Modules

Lorsqu'un ViewModel récupère des données à partir d'une ou plusieurs sources externes (telles que des bases de données ou des services web), il est important de séparer le code lié à la gestion de ces sources de données de la classe ViewModel. Ne pas le faire violerait, en fin de compte, les directives de séparation des responsabilités. Pour éviter de mélanger cette fonctionnalité avec le ViewModel, les directives architecturales de Google recommandent de placer ce code dans un module Repository séparé. Un repository n'est pas un composant d'architecture Android, mais plutôt une classe Java créée par le développeur de l'application qui est responsable de l'interaction avec les différentes sources de données. Cette classe fournit ensuite une interface au ViewModel permettant de stocker ces données dans le modèle.

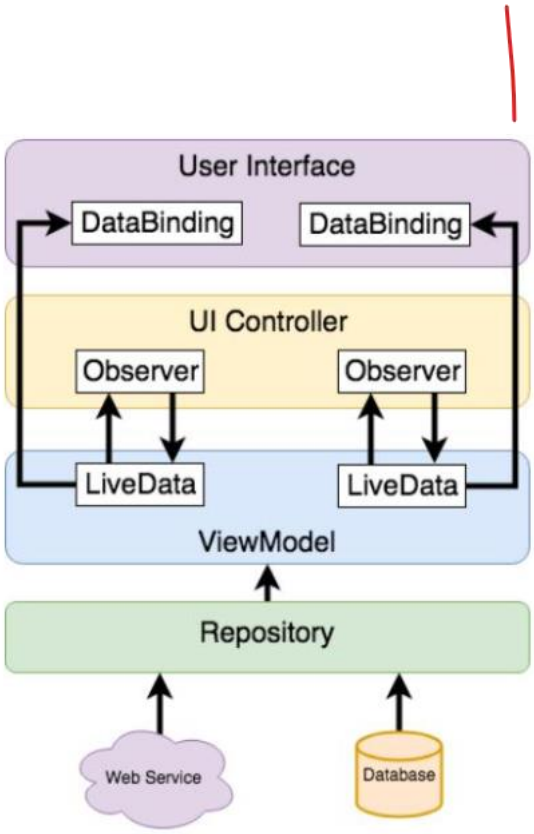

Figure 8 Repository Module

9.8 Summary

Jusqu'à l'année dernière, Google ne recommandait pas spécifiquement d'approche pour structurer une application Android. Cependant, cela a changé avec l'introduction d'Android Jetpack, qui comprend un ensemble d'outils, de composants, de bibliothèques et de directives d'architecture. Google recommande maintenant de diviser un projet d'application en modules distincts, chacun étant responsable d'une fonctionnalité spécifique, également appelée "séparation des préoccupations". En particulier, les directives recommandent de séparer le modèle de données de la vue de l'application du code chargé de gérer l'interface utilisateur. De plus, le code responsable de la collecte de données à partir de sources telles que des services web ou des bases de données devrait être regroupé dans un module de repository distinct plutôt que d'être inclus dans le modèle de vue.Android Jetpack comprend les Android Architecture Components, qui ont été spécialement conçus pour faciliter le développement d'applications conformes aux directives recommandées. Ce chapitre a introduit les composants ViewModel, LiveData et Lifecycle, qui seront abordés plus en détail dans le prochain chapitre. D'autres composants d'architecture non mentionnés dans ce chapitre seront couverts ultérieurement dans le livre.

10 The Android Room Persistence Librairy

10.1 Introduction

La bibliothèque de persistance Room, incluse dans les Android Architecture Components, est spécialement conçue pour faciliter l'ajout de la prise en charge du stockage de base de données aux applications Android, de manière cohérente avec les directives d'architecture Android. Ce chapitre explorera les concepts de base de la gestion de base de données basée sur Room, les éléments clés

qui travaillent ensemble pour implémenter la prise en charge de Room dans une application Android, (et comment cela est mis en œuvre en termes d'architecture et de codage => dans le livre).

10.2 Revisiting Modern App Architecture

Le chapitre intitulé "Architecture moderne des applications Android avec Jetpack" a introduit le concept d'architecture moderne des applications et souligné l'importance de la séparation des différentes responsabilités au sein d'une application. Le schéma illustré dans la Figure 9 présente l'architecture recommandée pour une application Android typique.

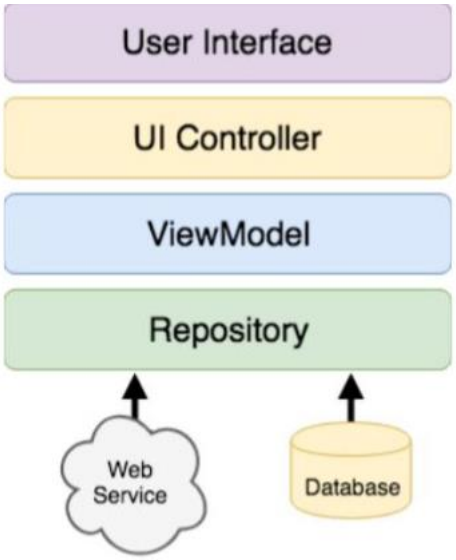

Figure 9 Modern App Architecture

10.3 Key Elements of Room Database Persistence

Avant d'entrer dans les détails plus tard dans le chapitre, il est tout d'abord utile de résumer les éléments clés impliqués dans le travail avec les bases de données SQLite en utilisant la bibliothèque de persistance Room.

10.3.1 Repository

Comme mentionné précédemment, le module de répertoire contient tout le code nécessaire pour gérer directement toutes les sources de données utilisées par l'application. Cela évite au contrôleur de l'interface utilisateur et au ViewModel de contenir du code qui accède directement aux sources telles que les bases de données ou les services web.

10.3.2 Room Database

L'objet de base de données Room fournit l'interface vers la base de données SQLite sous-jacente. Il fournit également au répertoire l'accès à l'objet d'accès aux données (DAO). Une application ne devrait avoir qu'une seule instance de base de données Room, qui peut ensuite être utilisée pour accéder à plusieurs tables de base de données.

10.3.3 Data Access Object (DAO)

Le DAO contient les instructions SQL requises par le répertoire pour insérer, récupérer et supprimer des données dans la base de données SQLite. Ces instructions SQL sont associées à des méthodes qui sont ensuite appelées depuis le répertoire pour exécuter la requête correspondante.

10.3.4 Entities

Une entité est une classe qui définit le schéma d'une table dans la base de données. Elle spécifie le nom de la table, les noms et types de colonnes, ainsi que la clé primaire. En plus de déclarer le schéma de la table, les classes entité contiennent également des méthodes getter et setter qui permettent d'accéder à ces champs de données. Les données renvoyées au répertoire par le DAO en réponse aux appels de méthodes de requête SQL prendront la forme d'instances de ces classes entité. Les méthodes getter seront ensuite appelées pour extraire les données de l'objet entité. De même, lorsque le répertoire doit écrire de nouveaux enregistrements dans la base de données, il créera une instance d'entité, configurera les valeurs de l'objet via des appels de setters, puis appellera les méthodes d'insertion déclarées dans le DAO, en passant par les instances d'entité à enregistrer.

10.3.5 SQLite Database

La base de données SQLite réelle est responsable du stockage et de l'accès aux données. Le code de l'application, y compris le répertoire, ne doit jamais accéder directement à cette base de données sous-jacente. Toutes les opérations de la base de données sont effectuées en utilisant une combinaison de la base de données Room, des DAO et des entités. Le diagramme d'architecture illustré dans la figure 10 montre comment ces différents éléments interagissent pour fournir un stockage de base de données basé sur Room dans une application Android.

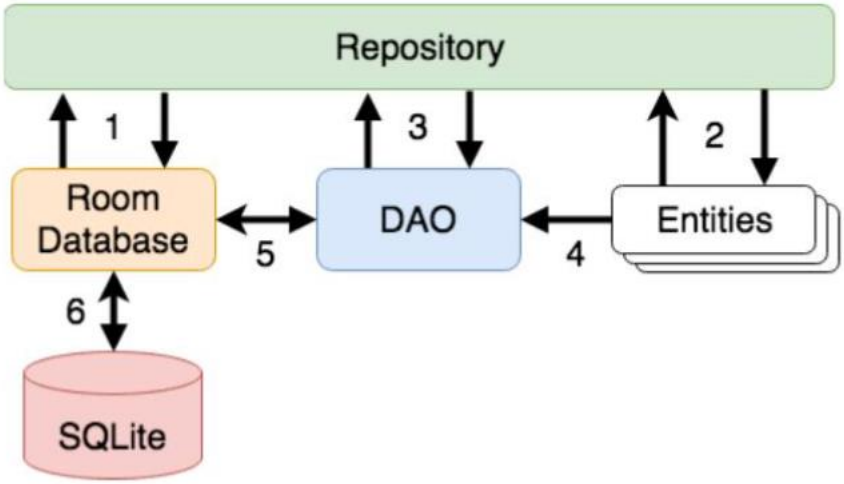

Figure 10 SQLite Database

Le schéma d'architecture ci-dessus peut être résumé comme suit :

1. Le répertoire interagit avec la base de données Room pour obtenir une instance de la base de données, qui est ensuite utilisée pour obtenir des références aux instances DAO.

2. Le répertoire crée des instances d'entités et les configure avec des données avant de les passer au DAO pour les opérations de recherche et d'insertion.

3. Le répertoire appelle des méthodes sur le DAO en passant des entités à insérer dans la base de données et reçoit des instances d'entités en réponse aux requêtes de recherche.

4. Lorsqu'un DAO a des résultats à retourner au répertoire, il les regroupe dans des objets d'entité.

5. Le DAO interagit avec la base de données Room pour initier les opérations de base de données et gérer les résultats.

6. La base de données Room gère toutes les interactions de bas niveau avec la base de données SQLite sous-jacente, soumettant des requêtes et recevant des résultats.