/home/fdambrin

Zeste de savoir

Développer et publier une app Android Material Design

5 janvier 2018

Table des matières

1. Introduction

Nous présenterons dans ce tutoriel comment développer et publier une application Android Material Design de A à Z. Pour ceux qui ne connaissent pas le Material Design, sachez qu'il s'agit d'un design conçu par Google avec la venue de la version Lollipop d'Android. Ce design aux apparences simplistes et aux nombreux points communs avec le Flat Design est bien plus complexe qu'il n'y parait. A partir du second chapitre, vous en saurez déjà plus !

Sachez aussi que toutes les sections qui constituent ce tutoriel disposent d'une version vidéo hébergée sur YouTube. Ces vidéos abordent les sujets de chacune de ces sections dans un face à face entre vous et moi. D'abord, je vous introduirai le sujet avec un décor bien sympathique, puis on terminera sur des lives où vous pourrez concrètement voir comment fonctionne les choses.

Cependant, les sujets abordés ne sont pas forcément adéquats pour les débutants. Les deux premiers chapitres sont accessibles pour tous mais à partir du troisième chapitre, il faut quelques légers pré-requis avant de se lancer dans la lecture :

1. Développement Android oblige, vous devez avoir des connaissances en Java et savoir manipuler des fichiers XML. C'est la base des connaissances pour le développement Android. Si vous n'êtes pas à ce niveau, il est sans doute prématuré de vous lancer dans ce tutoriel. Commencez par les deux premières parties du tutoriel sur le Java , puis enchainez par le tutoriel Android pour les débutants .
2. Si vous savez manipuler les menus, c'est un plus non négligeable. Le composant pour intégrer un menu latéral utilise massivement ce concept et il en est de même pour la nouvelle `Toolbar`. Ce n'est pas obligatoire mais si vous voulez être bien préparé et que vous ne connaissez pas ce concept, lisez ce chapitre du tutoriel Android pour les débutants.

> Tous les codes sources exposés dans ce tutoriel sont issus d'un projet open source . Ce projet regroupe des exemples de plusieurs tutoriels, la partie consacrée à notre tutoriel ci-présent se trouve à cette URL .

La rédaction de ce tutoriel n'est pas terminée. Il manque quelques chapitres mais le tutoriel en l'état vous permettra déjà de vous enseigner les bases du Material Design, vous expliquera tout ce qu'il y a à savoir sur les outils de développement et vous fera développer un premier composant très intéressant, le menu latéral ! Quant aux vidéos, la fréquence de publication est différente par rapport à la version écrite mais puisqu'elle ne nécessite aucune validation de la part de Zeste de Savoir, toutes les vidéos devraient rapidement être intégrées !

2. Vos premiers pas avec Android Studio

Depuis la venue d'Android Studio, cet environnement de travail s'impose et séduit les développeurs Android par ses fonctionnalités, sa rapidité et son socle logiciel solide. Depuis sa version 1, sortie le 8 décembre 2014, Google le présente comme l'IDE (pour *Integrated Development Environment*) officiel dans le développement Android. Ni les débutants, ni les professionnels peuvent faire l'impasse sur son utilisation.

Les explications données sont, au plus possible, accessibles à toute personne peu importe son niveau. Par contre, ce chapitre n'aborde pas toutes les manipulations et les fonctionnalités de ce logiciel tant elles sont nombreuses. Le socle logiciel d'Android Studio est l'IDE IntelliJ IDEA et il bénéficie alors de toutes ses fonctionnalités. Seulement les fonctionnalités les plus importantes et notables en dehors de celles d'IntelliJ IDEA sont abordées. De ce fait, disposer d'une expérience avec IntelliJ IDEA vous aidera dans la prise en main d'Android Studio mais ne vous pénalisera pas dans le cas contraire.

Ce chapitre se sépare en différentes sections. La section 1 explique la création des différents projets Android. Rappelez-vous que Android ne se limite plus aux téléphones et aux tablettes. Aujourd'hui, des voitures, des télévisions ou même des montres tournent sur ce système. La section 2 touche quelques mots sur le moteur de production Gradle mais les explications sont modestes au vu des innombrables fonctionnalités de cette technologie. Pour finir, la section 3 aborde le confort d'utilisation du logiciel tant au niveau des ressources XML que dans le code source Java.

Le développement d'Android Studio est encore très actif. Prenez donc note que toutes les explicitations données dans les sections à venir ont été rédigés avec sa version 1.3. Certaines des manipulations expliquées peuvent différer si vous utilisez une version plus récente qui serait sortie après la rédaction de ce chapitre. Si vous détectez des différences dans les captures d'écran ou

dans les manipulations exposées, n'hésitez pas à me contacter par messagerie privée.

2.1. Créer un projet Android

Si vous installez Android Studio pour la première fois, vous passerez par quelques écrans d'installation pour le configurer. Durant ces étapes préliminaires, pensez simplement à spécifier la localisation de votre SDK Android si vous en disposez déjà d'un sur votre poste de travail. Il serait dommage d'en installer un autre à un endroit différent au vu de la place qu'il peut prendre (peut dépasser les 20 Gigas).

Par contre, si c'est une première expérience pour vous le développement Android, choisissez l'installation standard dans cet enchainement d'écran pour installer le SDK Android au passage. Android Studio vous installera le logiciel et le SDK avec le support de la dernière version Android.

Pour rappel, ce logiciel est disponible sur cette page de la documentation officielle et disponible sur toutes les plateformes courantes, à savoir Windows, Linux et OS X.

FIGURE 2.1. – Message de bienvenue de la part d'Android Studio

L'installation terminée, vous pouvez créer votre premier projet sous Android Studio. Lancez la procédure de création d'un projet pour démarrer un nouvel enchainement d'écran où nous spécifierons toutes les informations nécessaires au projet et les plateformes Android qu'il supportera.

FIGURE 2.2. – Configurez votre nouveau projet Android avec son nom et sa localisation

Rappelez-vous que toutes les sources exposées dans ce tutoriel sont disponibles sur ce projet GitHub . Il n'y a aucune obligation de votre part de suivre à la lettre les manipulations données mais puisque le projet LearnIt sert de référence

aussi bien les sources que les informations sur le projet sont en accord avec cette référence.

Donc, sur notre premier écran, pour la configuration d'un nouveau projet, nous renseignons "LearnIt" comme nom d'application. Ce nom sera utilisé pour le projet sur votre ordinateur et Android Studio va créer une nouvelle ressource qu'il liera avec le nom de l'application pour l'afficher sur les terminaux Android (réel ou non).

Si vous êtes un développeur Android, vous connaissez l'importance du nom de domaine de la compagnie. Pour les débutants, voici les explications. En fait, plusieurs applications peuvent avoir le même nom. Il peut alors y avoir un problème de différenciation entre une application LearnIt provenant d'un développeur A et une autre application LearnIt d'un développeur B. Google demande alors de spécifier un nom de domaine unique pour pouvoir identifier ces 2 applications.

Dans cet exemple, le nom de domaine spécifié est "randoomz.org". Il est vivement conseillé de ne pas l'utiliser pour vous si vous comptez publier votre application Material Design sur le Play Store Android. Ce nom de domaine est déjà utilisé et vous serez bloqué à l'étape de publication. Cela dit, ce nom de domaine n'est pas obligé d'être un nom de domaine réel. Soit vous avez déjà un nom de domaine et dans ce cas, utilisez le ici. Sinon, inventez en un nouveau que vous utiliserez pour toutes vos applications. Si à l'étape de publication, le nom de domaine est utilisé, pas de panique, il sera encore temps de le modifier.

Pour finir, spécifier la localisation du projet. Par convention, créez votre projet dans un dossier nommé "workspace". C'est une convention un peu vieille mais qui permet de ne pas mélanger vos documents avec vos projets. Puis, spécifiez à nouveau le nom du projet pour que Android Studio crée tous les fichiers nécessaires dans ce même répertoire.

FIGURE 2.3. – Sélection des plateformes Android sur lesquelles l'application pourra s'exécuter

Spécifions maintenant les plateformes Android avec lequel l'application sera compatible. Malheureusement, ce tutoriel n'aborde pas le développement sur les montres, les télévisions, les voitures, ni les Google Glass (oui, ce produit existe toujours !). Contentez-vous de cocher "Phone and Tablet". Le reste des plateformes viendront avec d'autres tutoriels. En attendant, la documentation aborde le développement sur chacune de ses plateformes, il suffit de choisir son guide .

FIGURE 2.4. – Sélection d'une Activity par défaut pour commencer le projet

Depuis 2009, Google a bien compris que le développement Android pouvait être répétitif sur certains aspects. Globalement, à chaque création de projet, il était courant de développer un écran précédemment développé dans un autre projet. Par exemple, il serait légitime de développer un premier écran d'identification pour des applications mobiles de Zeste de Savoir, GitHub ou tous les autres sites avec un processus d'identification. Dans ce cas, Android Studio propose un canevas nommé "Login Activity" où il génèrera toutes les classes et les ressources nécessaires à la création de cet écran. C'est une fonctionnalité pratique et appréciée des développeurs.

Pour cet exemple, nous créons un écran simple grâce à l'option "Blank Activity". Certes, c'est moins fun mais pour des questions d'apprentissages, c'est mieux de commencer de zéro plutôt que d'avoir une tonne de truc générés dès le départ que vous ne comprendrez pas. Sachez quand même qu'il sera possible de générer tous ces canvas à tout moment durant le développement de votre application. Donc si vous ne faites pas joujou avec eux maintenant, il sera possible de le faire plus tard. La procédure sera expliquée dans une prochaine section de ce chapitre.

FIGURE 2.5. – Personnalisation des informations sur l'Activity choisie précédemment

Dernière étape, la personnalisation de l'écran choisi. Dans notre cas, la personnalisation est simple puisqu'il n'y a qu'une `Activity` et ses ressources. Si vous avez opté pour une autre `Activity`, aucune inquiétude. Les valeurs par défaut remplies par Android Studio suivent des standards qu'il est bon de suivre. D'ailleurs, nous ne modifierons rien sur cet écran dans le cadre de ce tutoriel.

— "Activity Name" est le nom de la classe qui étendra `Activity`. Par convention, elle a l'habitude de se nommer `MainActivity` pour la qualifier d'écran principal.
— "Layout Name" est le nom de la vue associée à l'écran sous la forme d'un fichier XML. La convention voudrait de convertir le nom de l'écran associé avec des *underscore* en renseignant le type du contrôleur avant (`Activity`, `Fragment` ou `View`).

— "Title" est le nom de l'écran. Une ressource sera créée et sera utilisée dans l'Android Manifest pour spécifier à Android que le nom de cet écran sera celui indiqué dans la ressource créée.
— "Menu resource name" est le nom de la ressource du menu de cet écran. Même si vous n'avez pas besoin d'un menu, Android Studio vous en créera un. Si vous voulez le retirer, vous devrez le faire à posteriori en supprimer le fichier XML et en retirer le code Java associé au menu dans l'`Activity` créé. Rien de bien compliqué et on notera la possibilité de pouvoir indiquer à Android Studio de ne rien générer du tout.

2.2. Le moteur de production Gradle

Cette section est difficile à expliquer. Gradle est un outil ultra puisant qui vous permettra de faire des tonnes de choses. Contrairement à Maven, qui est souvent perçu comme son prédécesseur bien que loin d'être enterré, Gradle n'est pas qu'un simple gestionnaire de dépendances mais un moteur entier de production. Grâce au Groovy, langage utilisé dans les scripts Gradle, vous pourrez exécuter des tâches complexes qui ne sont pas toujours liées directement à votre projet comme effectuer des commandes Git ou consolider un certain nombre de ressources externes pour votre projet.

Vous conviendrez qu'il serait utopique de vous expliquer dans son intégralité même les bases de Gradle. D'ailleurs, sa documentation officielle est juste gigantesque, de quoi décourager pas mal de développeurs, et aborde peu, voire pas du tout, les instructions spécifiques à Android alors que c'est cette même plateforme qui a contribué fortement à son utilisation au delà des simples projets Android. Aujourd'hui, beaucoup de projets Java, Groovy, C, C++, etc. utilisent Gradle.

Alors pourquoi Android utilise Gradle ? Et qu'est ce que nous allons aborder dans cette section ? Pour répondre à la première question, de mon propre ressenti, Google devait se sentir à l'étroit dans Maven. Pour avoir connu l'avant Gradle, il était souvent pénible de gérer ses dépendances, de mettre en production une application ou même d'insérer une dépendance Android dans Maven pour avoir un environnement de travail confortable. Bien que complexe, Gradle reste une alternative sérieuse et s'intègre bien mieux dans l'eco-système. Pour répondre à la seconde question, nous aborderons uniquement ce qui est spécifique à Android tout en expliquant certaines notions de base pour savoir ce que vous faites. Inutile de copier/coller bêtement ce qui est écrit dans ce tutoriel. Sinon, il suffit de se rendre sur les sources de ce tutoriel et d'en copier le contenu dans votre projet en modifiant ce qui vous plait.

2.2.1. Hiérarchie d'un projet

Avant de parler de Gradle, parlons de la hiérarchie d'un projet Android avec Gradle, de ses différents répertoires et fichiers. Le projet standard est souvent multi-module, c'est-à-dire que sa racine ne contient pas de sources mais des modules qui contiennent ces sources. Dans le cadre des sources de ce tutoriel, un seul module est créé et, est nommé "app" par convention. Ce module contient toutes les sources de l'application Android et toutes ses ressources.

A quoi pourrait correspondre d'autres modules ? Par exemple, si vous avez décidé de créer une version *wearable* (adapté pour les montres) de votre application, un nouveau module est nécessaire pour contenir toutes les sources de cette application. Puis, si cette application partage du code commun avec l'application téléphone, un troisième module aurait été une bonne solution pour contenir toutes ces sources et figurer comme dépendance au module *wearable* et du téléphone. Un module peut donc être une application à part entière ou une bibliothèque.

Toujours à la racine du projet, une série de répertoires et de fichiers sont exclusivement pour Gradle. Ce sont tous des fichiers standards qui sont utilisés dans n'importe quel projet Gradle, Android ou non. Il y a le répertoire `gradle` et les fichiers `gradle.properties`, `gradlew` et `gradlew.bat`. Le "w" à la fin des fichiers signifiant *wrapper* . En fait, ils aident à l'utilisation de Gradle au sein de votre environnement de développement et installent un environnement commun sur tous les intervenants d'un projet. C'est ainsi qu'un projet Gradle rendu open-source à travers une plateforme comme GitHub rend la prise en main et la collaboration plus aisée. Dans le cadre de ce tutoriel, vous verrez qu'on utilisera ce wrapper en ligne de commande. Cela sera expliqué par la suite.

Pour finir, nous avons les fichiers gradle. Il y a `build.gradle` et `settings.gradle` a la racine du projet et un `build.gradle` à la racine de chaque module. Dans le fichier `settings.gradle`, vous y trouvez la spécification de tous les modules du projet. Vous devriez voir simplement la ligne `include ':app'` puisque nous n'avons qu'un seul module. Si nous en créons un nouveau avec comme nom "lib", nous aurons alors `include ':app', ':lib'`. Vous pouvez créer des modules à la main et modifier ce fichier mais sachez qu'Android Studio permet aisément la création d'un module dans un projet. Pour ce faire, utilisez le menu `File > New > New Module`. Après avoir renseigné les informations au sujet du module, Android Studio se chargera du reste.

Quant aux fichiers `build.gradle`, ils seront expliqués juste après. En fait, ils sont plus ou moins liés (du moins, du fils vers le parent) et définissent toute la

configuration de construction du projet. Ce sont dans ces fichiers que vous spécifiez les dépendances, les *plugins* et la configuration Android de vos modules.

2.2.2. Configuration top-level Android

Les fichiers `build.gradle` spécifient toutes les informations concernant la configuration Android et peuvent vite devenir complexes. Par exemple, nous aurions pu parler des variantes, grâce aux *flavors*, une fonctionnalité typique du *plugin* Android et qui manque terriblement aux projets Java standard. Malheureusement, ce concept est un peu trop avancé pour ce tutoriel mais pourrait faire l'objet d'un autre tutoriel. Dans la plupart des cas, vous n'aurez pas besoin d'éditer le fichier *build* à la racine du projet, seulement ceux des modules. Mais il est utile de savoir comment ils fonctionnent pour mieux pouvoir éditer ceux des modules. Celui de *LearnIt* ressemble à ceci :

[Configuration racine du projet Gradle Android]groovy // Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript repositories jcenter() dependencies classpath 'com.android.tools.build :gradle :1.2.3' classpath 'com.neenbedankt.gradle.plugins :android-apt :1.4'

// NOTE : Do not place your application dependencies here ; they belong // in the individual module build.gradle files

allprojects repositories jcenter()

Qu'avons-nous là ? Dans `allprojects { ... }`, nous définissons tout ce qui est en commun à tous les modules de votre projet. Vous remarquez que les modules sont tellement souvent des applications que l'instruction est nommé "Tous les projets" et non pas "Tous les modules". La seule chose que nous spécifions commune à tous les modules est le dépôt où Gradle va chercher les dépendances que nous spécifions.

Alors pour ceux qui ne connaissent pas les gestionnaires de dépendances comme Maven, voici une brève explication pour savoir comment cela fonctionne : il existe sur internet de nombreux dépôts publics ou privés qui hébergent les binaires de bibliothèques. Tout le monde peut créer de nouveaux dépôts ou utiliser les dépôts existants. Cependant, il y a des dépôts plus utilisés que d'autres. Par exemple, Maven Central est sans doute l'un des plus gros dépôts Maven. Il regroupe des milliers de binaire qui sont en accès libre et tout le monde peut y placer ses propres binaires (sous demande). Dans vos projets Maven ou Gradle, il suffit alors de spécifier une dépendance vers l'un de ces binaires pour les télécharger sur votre machine de développement et l'utiliser. Les outils comme Maven et

Gradle s'occupent eux-même de rajouter ces dépendances dans le *classpath* de votre projet.

> **Note :** Le *classpath* est un paramètre passé à la JVM qui définit le chemin d'accès au repertoire où se trouvent les classes Java et les binaire afin qu'elle les exécute.

Revenons à la configuration commune à tous les modules, son contenu doit être plus clair. Nous spécifions simplement à Gradle qu'il doit chercher les dépendances dans le dépôt jcenter , une alternative à Maven Central.

Dans `buildscript { ... }`, nous configurons les *classpaths* nécessaires aux scripts Gradle des sous modules. Premièrement, nous spécifions le dépôt Maven des dépendances *buildscript*. Puis, nous spécifions les *classpaths* voulues.

— `classpath 'com.android.tools.build:gradle:1.2.3'` est un *classpath* essentiel. C'est le *plugin* Android Gradle qui permet la configuration des modules du projet. Il est généré automatiquement à la création du projet.
— `classpath 'com.neenbedankt.gradle.plugins:android-apt:1.4'` a été rajouté pour un autre *plugin*. Il sera expliqué plus tard mais sachez que vous en avez pas forcément besoin. D'ailleurs, il est plus probable que vous n'en aurez jamais besoin.

2.2.3. Configuration des modules

Dans la section précédente, nous avons configuré tout ce qui était commun aux modules. Tous les plugins utilisés et les dépendances renseignées sont disponibles grâce à la configuration de ce fichier *build* à la racine du projet. Sauf si vous désirez renseigner des dépendances spécifiques à un module précis, vous ne renseignerez pas de dépôt dans les fichiers *build* des modules. Voici un exemple tiré du projet de ce tutoriel :

[Configuration d'un module Gradle Android]groovy apply plugin : 'com.android.application' apply plugin : 'com.neenbedankt.android-apt'

android compileSdkVersion 22 buildToolsVersion "22.0.1"

defaultConfig applicationId "org.randoomz.learnit" minSdkVersion 15 targetSdkVersion 22 versionCode 1 versionName "1.0" buildTypes release minifyEnabled false proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

dependencies compile 'com.android.support :support-v4 :22.2.0' compile 'com.android.support :appcompat-v7 :22.2.0' compile 'com.android.support :design :22.2.0' compile 'com.android.support :recyclerview-v7 :22.2.0' compile 'com.android.support :cardview-v7 :22.2.0' compile 'com.daimajia.swipelayout :library :1.2.0@aar'

compile 'com.squareup.dagger :dagger :1.2.2' apt 'com.squareup.dagger :dagger-compiler :1.2.2'

compile 'com.jakewharton :butterknife :6.1.0' compile 'com.jakewharton.timber :timber :3.0.2'

compile 'io.reactivex :rxjava :1.0.10' compile 'io.reactivex :rxandroid :0.24.0'

compile 'com.github.frankiesardo :auto-parcel :0.3' apt 'com.github.frankiesardo :auto-parcel-processor :0.3'

compile 'com.squareup.sqlbrite :sqlbrite :0.1.0' compile 'de.psdev.licensesdialog :licensesdialog :1.7.0'

Qu'avons-nous là ? D'abord, nous renseignons les *plugins* que nous voulons utiliser. Le *plugin* `com.android.application` est obligatoire dans les applications Android. C'est le *plugin* Gradle Android qui permet d'utiliser `android { ... }` et configurer votre application ou votre bibliothèque Android. Le *plugin* `com.neenbedankt.android-apt` est optionnel. Il est utilisé pour rajouter l'instruction `apt` dans les dépendances. Il ne sera pas expliqué dans le cadre de ce tutoriel. Pour avoir des informations supplémentaires au sujet de cette bibliothèque, n'hésitez pas à visiter son projet .

Une fois les *plugins* renseignés, vous pouvez enfin configurer votre projet Android ! Dès lors, il y a une chose importante à savoir. Avant Android Studio, et donc avant les projets Gradle, l'Android Manifest faisait foi. Toutes les informations renseignées dedans étaient la seule et quasi unique configuration de votre projet Android. Aujourd'hui, tout ce qui est renseigné dans `android { ... }` va écraser les données renseignées dans l'Android Manifest. Cela veut dire que si vous renseignez un numéro de version d'application différent dans l'Android Manifest et dans le fichier *build*, le *plugin* Gradle va écraser la valeur dans le *manifest* avec la valeur renseignée dans le *build*. Si vous êtes un ancien développeur Android, ne vous faites pas avoir !

> **Rappel :** Les *plugins* renseignés sont disponibles grâce aux *classpaths* dans le fichier *build* à la racine du projet.

Les informations renseignées dans le bloc `android { ... }` sont simples à comprendre :

— La propriété `compileSdkVersion` spécifie la version de l'API Android que vous utiliserez pour compiler votre application.
— La propriété `buildToolsVersion` spécifie la version du *Android SDK Build-tools* souhaitée. Il est préférable de mettre à jour régulièrement cette information pour la dernière version installable depuis le *SDK Manager*.
— Le bloc `defaultConfig { ... }` spécifie la configuration que vous allez surcharger dans le fichier *manifest* de votre application. Vous ne pouvez pas y renseigner toutes les permissions nécessaires à votre application, ni les écrans à travers les activités. Par contre, vous devez renseigner l'identifiant unique de votre application (`applicationId`), la version minimale de l'API Android que vous supportez (`minSdkVersion`), la cible de l'API Android (`targetSdkVersion`), la version interne de votre application (`versionCode`) et sa version externe (`versionName`).
— Le bloc `buildTypes { ... }` spécifie comment compiler et empaqueter votre application dans différents modes, *release* ou *debug* par exemple. Sachant que dans notre exemple, nous laissons la configuration par défaut qui demande à votre environnement de travail de ne pas minifier le code source de l'application ni d'obfusquer l'application avec *Proguard*.

Nous appelons une version interne d'une application, le code qui est utilisé par le système Android et non visible à vos utilisateurs. Quant à la version externe, il s'agit du code visible auprès de vos utilisateurs soit dans le Play Store, soit dans les informations de l'application, soit dans votre application si vous souhaitez l'afficher. Cette dernière version peut prendre n'importe quelle valeur mais tentez de conserver une certaine convention dans son utilisation. Cela peut être une indication utile pour vous ou vos utilisateurs.

Nous retournons dans une configuration standard d'un projet Gradle avec le bloc `dependencies { ... }`. Comme son nom l'indique, vous y placerez toutes les dépendances de votre projet. Pour ceux qui n'utilisent pas encore un gestionnaire de dépendances, il n'est plus nécessaire de télécharger ses exécutables et les placer dans un dossier `libs`. Il suffit de connaitre l'identifiant unique de la bibliothèque (accessible via des moteurs de recherche comme celui de Maven Central) et de les placer dans ce bloc. Gradle se chargera de télécharger l'exécutable sur votre poste de travail et de l'inclure à la compilation de votre projet.

Bien entendu, si pour une raison X ou Y, vous devez absolument renseigner un exécutable dans votre projet ou, plus courant, spécifier les sources d'un autre module dans votre projet, Gradle vous permet de le faire :

[Possibilité de déclaration des dépendances]groovy dependencies // Module dependency compile project(" :lib")

// Remote binary dependency compile 'com.android.support :appcompat-v7 :19.0.1'

// Local binary dependency compile fileTree(dir : 'libs', include : ['*.jar'])

2.3. Confort d'utilisation

Android Studio apporte un confort d'utilisation non négligeable face à son concurrent direct, ADT Eclipse. En plus de bénéficier de toutes les fonctionnalités d'IntelliJ IDEA en terme de refactoring et de personnalisation, Android Studio facilite la création des ressources et des composants. Impossible de toutes les énumérer, mais certaines de ces fonctionnalités sont utiles à connaitre. Vous êtes susceptible de les utiliser, tout comme moi, à chaque fois que vous développerez avec cet IDE.

Tout développeur se doit de maitriser son environnement de travail. Dans ce tutoriel, vous allez connaitre l'existence de quelques fonctionnalités sympathiques mais je vous encourage vivement à l'explorer d'avantage et à connaitre les raccourcis pour refactoriser votre code. Vous allez fortement gagner en productivité.

2.3.1. La barre d'outils

FIGURE 2.6. – Toolbar d'Android Studio avec ses raccourcis spécifiques pour Android

La barre d'outils regroupe des raccourcis utiles que vous serez amené à utiliser régulièrement. L'encadré bleu n'est pas spécifique à Android Studio, il est présent aussi dans IntelliJ IDEA. Ces raccourcis permettent d'exécuter des configurations d'exécution que vous avez soigneusement préparé ou que Android Studio aura créé pour vous. Par exemple, Android Studio crée automatiquement la configuration pour exécuter votre application sur votre terminal ou votre émulateur. Le menu affiche toutes les configurations à exécuter, la flèche verte lance l'exécution et le petit insecte lance l'exécution en mode *debugger*.

Nous n'allons pas apprendre à utiliser le mode *debugger* dans ce tutoriel mais je voulais le mentionner pour que vous sachez qu'il existe. Bien avant que les IDE modernes existent, la manière la plus répandue de trouver des bugs dans son code était d'imprimer sur la sortie standard des données. Aujourd'hui, des *debuggers* existent dans tous les IDE et permettent d'arrêter l'exécution d'un programme à des points stratégiques que vous aurez indiqué à Android Studio. Je vous encourage vivement à en savoir plus sur ce mode via la documentation officielle de IntelliJ . La maitrise de ce mode vous fera grandement gagner en productivité et vous ne pourrez plus vous en passer après.

L'encadré rouge regroupe les seuls raccourcis spécifiques à Android Studio et donc à Android. Le premier en partant de la gauche force la re-synchronisation de vos dépendances au niveau Gradle. Le second lance *Android Virtual Device Manager* pour consulter tous vos émulateurs. Le troisième vous permet d'améliorer votre SDK en téléchargeant de nouvelles versions d'API Android ou des bibliothèques signées de Google. Pour finir, le dernier lance *Android Device Monitor* pour consulter des métriques sur votre application comme sa consommation Internet ou l'utilisation de la mémoire. Ces métriques sont poussées mais peuvent être utiles quand vous faites attention à la qualité de vos applications. Vous l'aurez compris, consultez-les de temps en temps et analysez-les pour améliorer votre code.

2.3.2. Onglet Project

FIGURE 2.7. – Passer de la vue Project à la vue Android

Situé à gauche du logiciel, un onglet "Project" permet la visualisation de tous les fichiers présents dans votre projet organisés selon la vue sélectionnée. Par défaut, la vue est "Project". Il en existe 8 différentes, à l'heure où sont écrites ces lignes, mais l'une d'elle est spécifique à Android Studio et a été conçue pour améliorer la gestion des fichiers des projets Android.

Sélectionnez la vue "Android" pour voir apparaitre tous vos modules, ici seulement *app*, et tous vos scripts Gradle. Dans chaque module, vous avez un accès simple et rapide à votre fichier *Android Manifest*, à vos sources Java et à vos ressources. Quant aux scripts Gradle, vous avez un accès direct à tous les fichiers *build*. Cette section regroupe toute la configuration de votre projet et de vos modules. N'hésitez

pas à consulter l'information entre parenthèse pour connaitre la provenance du fichier.

C'est maintenant à vous de décider de la vue que vous préférez !

FIGURE 2.8. – Créer une nouvelle ressource dans votre projet

Les plus anciens se souviendront du site Android Asset Studio . Un petit site développé par Roman Nurik, designer chez Google, qui facilitait considérablement la création de ressources dans toutes les tailles demandées par Android. Pour rappel, il est nécessaire de fournir une même ressource en plusieurs tailles pour que Android puisse récupérer la ressource adéquate en fonction de l'appareil sur lequel s'exécute l'application (smartphone, tablette, montre, etc.).

Avec Android Studio, vous restez dans votre environnement de travail. Pour créer une ressource dans toutes les tailles, cliquez droit sur le dossier *res*, puis `New > Image Asset` et une fenêtre s'ouvrira pour créer votre ressource.

FIGURE 2.9. – Créer de nouveaux composants dans votre projet

Pareil pour les composants du type activité, fragment ou des vues puisque vous pouvez générer des composants entiers. Par exemple, il arrive régulièrement d'avoir des écrans avec une liste dedans. Cliquez droit sur votre *package*, puis `New > Fragment > Fragment (List)` et Android Studio vous génère le fragment, son fichier XML et les autres ressources utiles si nécessaire.

2.3.3. Code source Java et ressources XML

FIGURE 2.10. – Consulter les fichiers XML attachés à une Activity, un Fragment ou une vue

A partir de n'importe quelle activité, fragment ou vue, vous pouvez accéder à tous les *layout* associés sur le côté gauche de votre code source au niveau de la déclaration de classe. Une petite icône, qui représente un fichier balisé, est cliquable et affiche une liste de tous ces fichiers XML. Cette petite colonne à gauche de votre

code source renferme pas mal d'autres raccourcis. Par exemple, si vous accédez à une chaine de caractères déclarée dans un fichier XML, un raccourci vous permettra de vous y rendre. Il en est de même pour les couleurs et d'autres ressources du même genre.

FIGURE 2.11. – Vue par défaut de vos interfaces dans votre fichier XML

Dernière fonctionnalité présente depuis les débuts d'Android Studio mais toujours aussi utile, c'est la prévisualisation de vos *layout*. Lorsque vous éditez un *layout*, un onglet à droite, nommé "Preview", permet la visualisation en direct de votre interface dans un terminal de votre choix, dans l'orientation de votre choix, avec le thème de votre choix, dans l'activité hôte de votre choix, dans la langue de votre choix et dans la version API Android de votre choix. Absolument tout est paramétrable et le rendu est le plus fidèle possible au rendu final sur un véritable téléphone.

Mais, cerise sur le gâteau, Android Studio permet aussi la prévisualisation sur plusieurs tailles différentes, plusieurs versions d'API Android différente et ainsi, en un coup d'oeil, détecter toutes les imperfections et/ou les incompatibilités de vos interfaces suivant tous ces paramètres. Cette prévisualiation est essentielle dans la confection de vos interfaces, abusez en jusqu'à obtenir quelque chose de correct sur le plus de téléphones différents.

FIGURE 2.12. – Les vues XML pour vos interfaces sont nombreuses et pleines de surprise !

3. En savoir plus sur le Material Design

Il se peut que vous connaissez déjà les principes du Material Design mais savez-vous comment réaliser des applications avec ce style ? Voire même, quelles sont ses bonnes pratiques ? Google a conçu un site entier sur ce sujet et vous allez vous rendre compte que c'est plus complexe qu'il n'y parait. Pour concevoir une application Material Design, il ne suffit pas de mettre des ombrages par ci et par là. Il y a des principes à connaitre, des règles à suivre et des pratiques à proscrire.

De formation, je suis ingénieur et non designer. Je n'ai donc pas la prétention de maitriser le Material Design. Tout ce qui est présenté dans cette section est un mélange d'expériences personnelles et de lecture de la documentation officielle des designers et des développeurs. De plus, il n'est abordé que les bases. Si vous êtes intéressé par cet aspect là du tutoriel, je ne peux que vous conseillez le site de Google consacré au Material Design.

3.1. Principes du Material Design

Lors de la conférence Google I/O 2014, Google a dévoilé le design d'Android Lollipop, design qui portera le nom de Material Design. Ce design met l'accent sur l'unification de l'interface entre les différents types d'appareils : téléphones, tablettes, montres, télévisions mais touche aussi les sites web. De nombreux sites, dont ceux de Google, utilisent le Material Design qui est tout à fait adapté à la pratique du responsive. Pratique qui consiste à adapter l'interface aux dimensions de la fenêtre du navigateur.

Le Material Design est apparu quand le Flat Design commençait à faire parler de lui et Google a été rapidement clair sur le sujet : le Material Design n'est pas du Flat Design. Les plus sceptiques sur cette affirmation diront alors que c'est le Flat Design à la Google. Ils n'auront que partiellement raison. Même s'il est vrai que le

Material Design partage un visuel similaire au Flat Design, il n'en dispose pas des principes forts du design fait par Google. Les designers de cette entreprise se sont mis au défi de créer un langage visuel pour leur utilisateurs qui se synthétisent en 3 grands principes.

3.1.1. Material est la métaphore

FIGURE 3.1. – Material est la métaphore

Une notion un poil complexe à comprendre, la métaphore est la théorie d'unification de la rationalisation de l'espace ou d'un système en mouvement. Pour ce faire, Google s'est inspiré du papier et de l'encre. Lorsque vous concevez vos interfaces, vous devez penser au comportement du papier. Par exemple, qu'est ce qu'il se passerait si vous les superposez avec des distances fixes entre elles.

Pour illustrer mes propos, nous allons nous rendre sur un petit site internet . Cette page web est issue d'un projet GitHub et développée par l'un des employés de Google, le designer Roman Nurik. Pour aider à la visualisation de cette superposition dans l'espace, vous pouvez faire pivoter des interfaces Material Design dans une vue 3D, et ainsi constater la conséquence de l'élévation des composants graphiques de votre application, c'est-à-dire des ombrages.

3.1.2. Audacieux, graphique et intentionnel

FIGURE 3.2. – Audacieux, graphique et intentionnel

Les éléments fondamentaux de l'impression, comme la typographie, l'espacement ou les couleurs, guide les traitements visuels. Ces éléments ont une incidence directe sur le plaisir des yeux puisqu'ils contribuent directement au design de votre application mais à une importance bien plus grande. Ils créent une hiérarchie, du sens et le focus sur une interface. Tous les composants graphiques, nécessitant une action de la part de l'utilisateur final, doivent attirer l'œil par une couleur, la taille de la typographie et son espacement. Tout cela crée une meilleure expérience utilisateur. Les fonctionnalités de base sont donc immédiatement apparentes et fournissent des points de validation visuelle pour l'utilisateur.

3.1.3. Le mouvement fournit du sens

FIGURE 3.3. – Le mouvement fournit du sens

Pour finir, et non des moindres, l'accent mis sur les animations. Les actions de l'utilisateur sont des points d'inflexion qui initient le mouvement et transforment l'ensemble de l'interface. Toutes les actions sont prisent dans un environnement unique. Les composants sont présentés à l'utilisateur sans casser la continuité de l'expérience même avec leurs transformations et leurs réorganisations.

Si nous prenons un exemple simple, nous disposons d'une application avec une liste, un en-tête et un bouton sur le côté de l'en-tête. Nous allons effectuer deux actions : exercer une pression sur un *item* de la liste, puis nous allons monter notre doigt pour descendre dans la liste. Qu'est ce qu'il va se passer ? Notre première interaction va lancer une première animation en sélectionnant l'*item* sur lequel le doigt est positionné. Ensuite, lorsque nous allons descendre dans la liste, l'en-tête va disparaitre vers le haut et le bouton va se réduire pour disparaitre. La raison est simple. Nous n'avons plus besoin d'afficher ces informations pour le moment alors le design nous les caches temporairement. Nous avons naviguer dans l'interface, chaque composant est dans un environnement unique mais tout est cohérent.

3.2. Propriétés du Material Design

3.2.1. Le rapport entre l'élévation et l'ombrage

L'une des propriétés les plus simples à comprendre mais peut-être la plus primordiale de toute, c'est le rapport qu'il y a entre l'élévation et l'ombrage d'un *materiel*[1]. Sachez que ce n'est pas le materiel ni dans sa forme, ni dans ses propriétés qui donne son ombre portée mais l'élévation qu'il a avec le *plan*[2].

Mais alors, qu'est ce qu'est l'élévation d'un matériel ? Imaginez un plan vide sur lequel nous allons rajouter un premier matériel. Ce matériel, nous allons l'**élever** d'un certain nombre pour qu'il prenne de la hauteur par rapport au plan. Disons que nous l'élevons à une valeur de 2dp[3]. Maintenant, rajoutons un second matériel où nous l'élevons à 8dp. Nous aurons alors un premier matériel avec une faible ombre portée et un second avec une ombre portée beaucoup plus importante.

FIGURE 3.4. – Différence entre 2 matériels d'une élévation différente

Vous comprendrez aisément le rapprochement avec le matériel réel. Si vous prenez deux feuilles de papier et que vous les positionnez de manière équivalente à notre configuration, vous obtiendrez le même résultat. Pour peu que votre source de lumière principale proviennent du plafond ! Mais les choses sont plus simples dans le développement Android, nous nous préoccupons pas de la source de lumière. Elle se positionne toujours au même endroit, c'est-à-dire au dessus de notre plan.

Les élévations sont personnalisables. Vous pouvez indiquer la valeur que vous souhaitez pour tous les composants graphiques. Mais vous n'êtes pas forcément maitre de tous les éléments de votre interface et des bonnes pratiques ont été élaborées pour rester cohérent avec l'élévation des composants Android. Par exemple, par défaut, la boite de dialogue a une élévation importante et la barre d'action plus réduite. Forcément, ces deux composants ne se retrouveront jamais au même niveau et donnent une impression de cohérence de votre interface.

FIGURE 3.5. – Élévation par défaut pour les composants Android les plus répandues

L'élévation par défaut n'est pas toujours automatiquement renseignée. Par exemple, pour le `Floating Action Button`, c'est de votre ressort d'appliquer l'élévation renseignée dans les bonnes pratiques du Material Design. Vous pouvez vous en apercevoir dans la figure précédente, la bonne pratique est de lui renseigner une élévation de 6 à son état stable et d'aller jusqu'à 12 lorsqu'il est à l'état pressé.

Vous l'avez peut-être remarqué, mais la `Snackbar` et le `Floating Action Button` possèdent une élévation de même valeur. Cela veut dire qu'il sont "physiquement" sur le même niveau par rapport au plan. Si nous étions dans le monde réel, il serait impossible de disposer deux composants exactement au même endroit dans l'espace, même s'ils se touchent, ils seront forcément à deux endroits différents. Dans le Material Design, c'est la même chose. Pour palier à ce problème, il est nécessaire de faire bouger ses éléments ! Si vous faites apparaitre une `Snackbar` en bas de votre écran, pensez à faire réagir votre bouton en conséquence s'il se trouve en bas. Heureusement, la bibliothèque prévoit ces animations usuelles et ces dernières sont facilement intégrables. Nous le verrons plus loin dans ce tutoriel.

Sur la figure ci-dessous, nous pouvons voir un exemple simple d'un design cohérent dans sa réalisation. Vous disposez de trois éléments : la barre d'action, de cartes qui

affichent de l'information et un bouton flottant en bas à droite de l'écran. Chaque composant suit sa valeur par défaut sur son élévation. La barre d'action est élevé à 4dp, le bouton flottant à 6dp et la carte à 2dp. Aucun composant n'est au même niveau et nous distinguons convenablement la position dans l'espace de chacun de ces composants.

FIGURE 3.6. – Un exemple simple d'élévation dans une application Android

Si vous voulez en savoir plus sur l'élévation des composants Android, allez lire la documentation à ce sujet du Material Design.

3.2.2. Les propriétés physiques

Puisque le matériel design s'inspire de la vie réelle, elle dispose de propriétés physiques qu'il est conseillé de suivre. D'ailleurs, réaliser certaines contre-propriétés serait difficilement réalisable et demanderait un effort particulier de votre part pour y parvenir. Si tel est le cas, retirez les mains de votre clavier et allez faire un tour. Vous aurez peut-être des meilleures idées après avoir respiré un bon coup !

Voici donc les priorités physiques de manière exhaustive :

Vous pouvez faire varier la hauteur et la largeur d'un matériel mais vous ne pouvez pas faire varier son épaisseur. Elle sera toujours de 1dp, dimension qui se rapproche le plus d'une feuille de papier.

FIGURE 3.7. – L'épaisseur est toujours à 1dp. Crédit : Google Design

Une ombre portée n'apparait pas si le matériel ne s'élève pas. C'est parce que le matériel s'élève qu'il fait apparaitre une ombre portée. Si vous êtes un utilisateur de logiciels comme Photoshop, oubliez les ombre portées que vous spécifiez sur n'importe quel élément. Quand vous concevez une interface Material Design, vous ne réfléchissez pas en ombre portée mais en élévation.

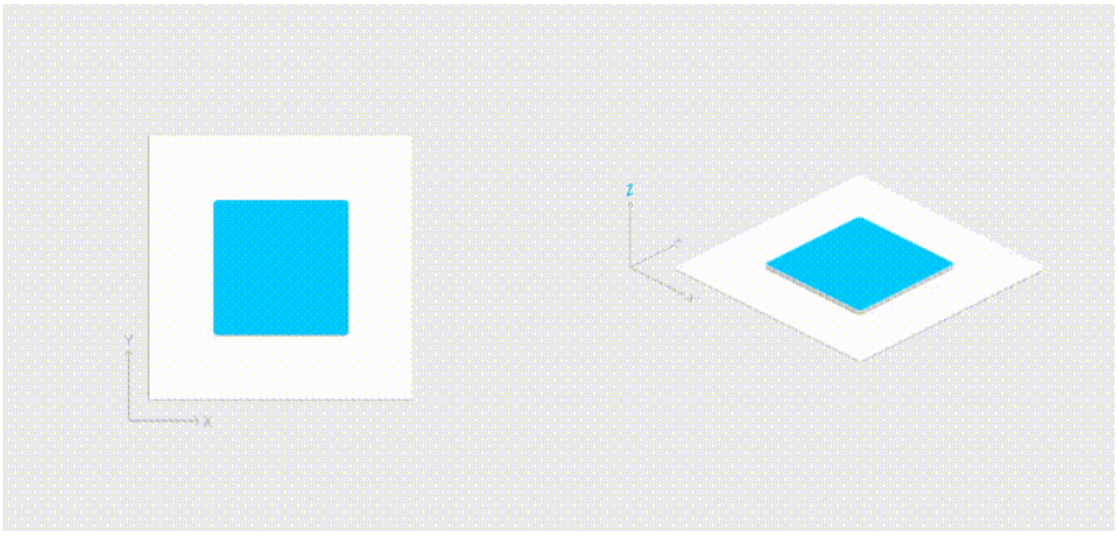


FIGURE 3.8. – L'ombrage est le résultat d'une élévation. Crédit : Google Design

Le contenu d'un matériel peut prendre toutes les couleurs et toutes les formes plates mais n'est pas sujet à l'élévation et ne dispose pas d'une épaisseur. C'est comme si vous dessinez sur une feuille de papier avec la possibilité de donner vie à ce dessin.

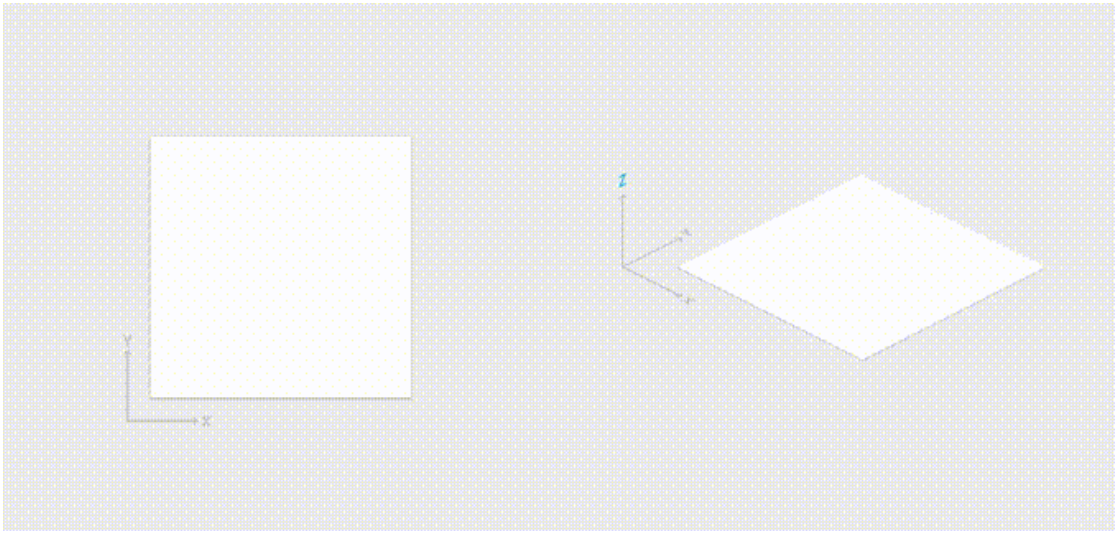


FIGURE 3.9. – Le contenu d'un matériel n'est pas sujet à l'élévation. Crédit : Google Design

Un contenu n'est pas forcément dépendant de son matériel. Il peut prendre qu'une partie de la surface, s'animer à l'intérieur mais il ne peut pas aller au delà des limites du matériel.

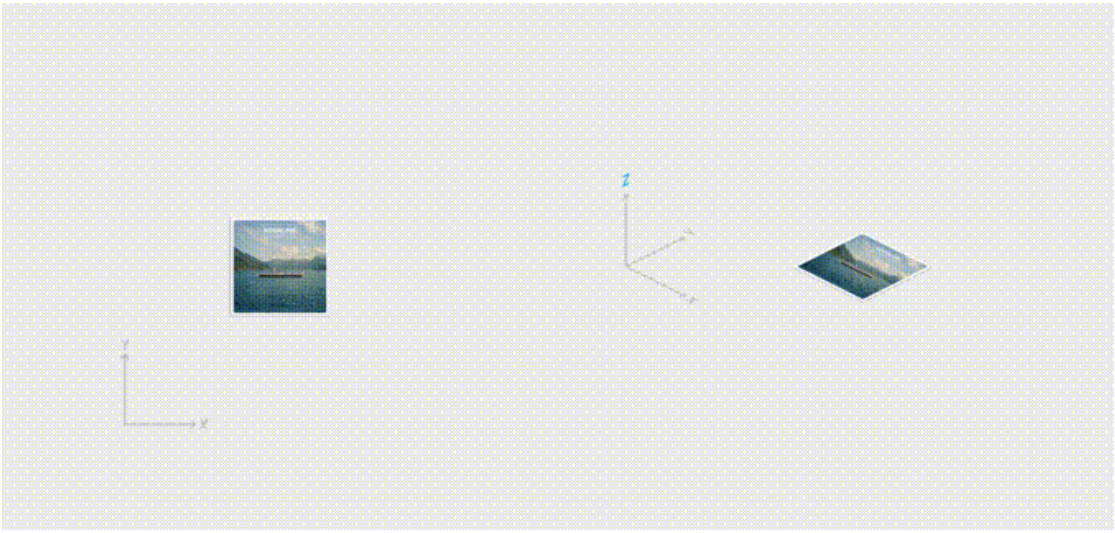


FIGURE 3.10. – Contenu limité par le matériel. Crédit : Google Design

La matériel est un élément solide. Vous ne pouvez pas traverser cet élément. Si vous touchez un composant, c'est ce composant qui réagit. Le composant en dessous ne réagit pas à l'interaction de l'utilisateur.

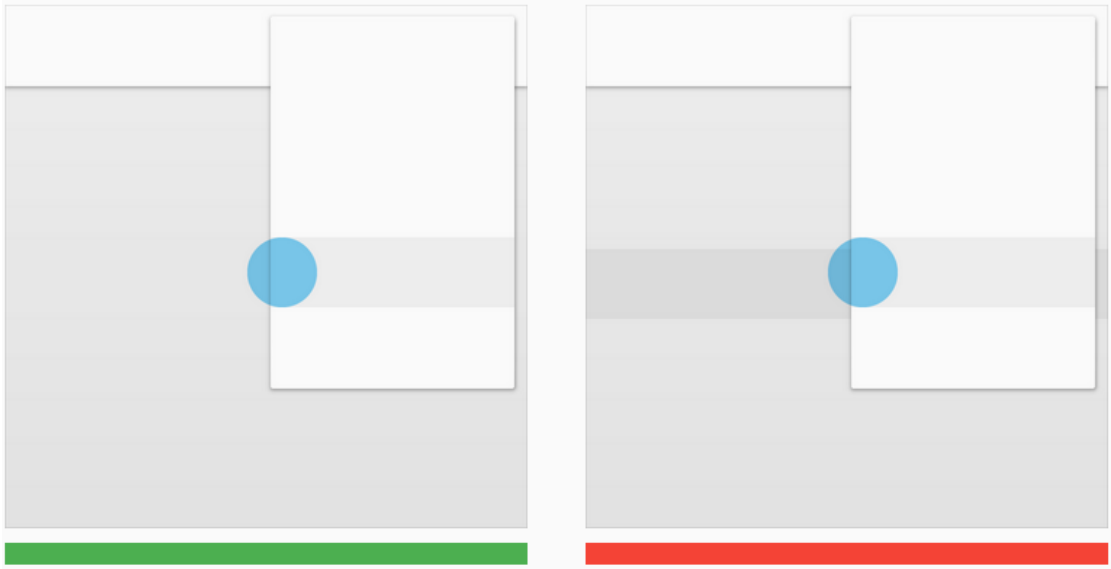


FIGURE 3.11. – L'interaction utilisateur ne traverse pas la matière. Crédit : Google Design

Deux matériels ne peuvent pas se retrouver à un même endroit dans l'espace, ils sont forcément à des élévations différentes ou à des endroits différents dans le plan. Nous rejoignions ici notre précédent exemple avec le bouton flottant qui réagit par rapport à un `Snackbar`. Puisqu'ils sont à une élévation identique, nous sommes obligés d'animer le bouton flottant par rapport à l'apparition de la `Snackbar`.

FIGURE 3.12. – Deux éléments ne peuvent pas se trouver au même endroit. Crédit : Google Design

Un matériel ne peut pas en traverser un autre. Si nous prenons un exemple concret dans la vie réelle, cela voudrait dire que vous pourriez traverser la matière, comme un mur. Bien que le pouvoir est *badass*, il n'en reste pas moins impossible à daté de ce jour.

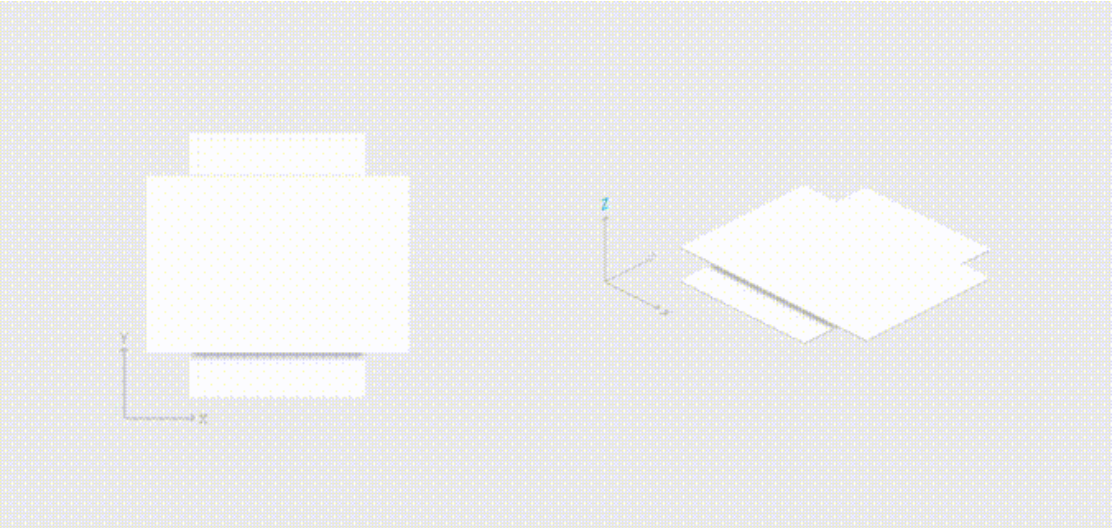


FIGURE 3.13. – Un matériel ne peut pas en traverser un autre. Crédit : Google Design

Un matériel peut changer sa forme et/ou sa taille. Un bouton flottant, initialement petit et rond, peut grandir et devenir un carré. L'inverse est possible aussi. Cette pratique dynamise vos interfaces et offre une cohérence à travers les actions.

FIGURE 3.14. – Le matériel peut changer de forme. Crédit : Google Design

Deux matériels de même taille peuvent fusionner et ne faire qu'un grand matériel. A partir de ce nouveau matériel, libre à vous de le réduire à la taille que vous le souhaitez ou de le diviser à nouveau. Pensez simplement à garder une cohérence dans les ombrages et dans la taille des matériels pour rester cohérent.

FIGURE 3.15. – La fusion entre les matériels est possible. Crédit : Google Design

Vous ne pouvez pas plier le matériel. Nous rejoignons l'effort particulier pour aller contre les bonnes pratiques du Material Design. Rien n'est prévu dans la bibliothèque design pour plier un matériel et vous allez dépenser du temps à le développer vous même. Optez plutôt pour un autre effet.

FIGURE 3.16. – Un matériel ne peut pas se plier. Crédit : Google Design

Un matériel peut être créé et détruit n'importe où. Faites simplement preuve de bon sens. Évitez la génération et la destruction de trop d'éléments à la fois au risque de rendre votre interface lourde à l'usage.

FIGURE 3.17. – Créer et détruire un matériel. Crédit : Google Design

Du moment qu'ils ne se trouvent pas au même endroit qu'un autre matériel, les matériels peuvent bouger sur tous les axes du plan. N'abusez pas des animations mais n'hésitez pas à utiliser cette translation quand c'est nécessaire.

FIGURE 3.18. – Un matériel peut se déplacer sur tous les axes. Crédit : Google Design

Le changement d'élévation sur un matériel est réservé à une interaction utilisateur.

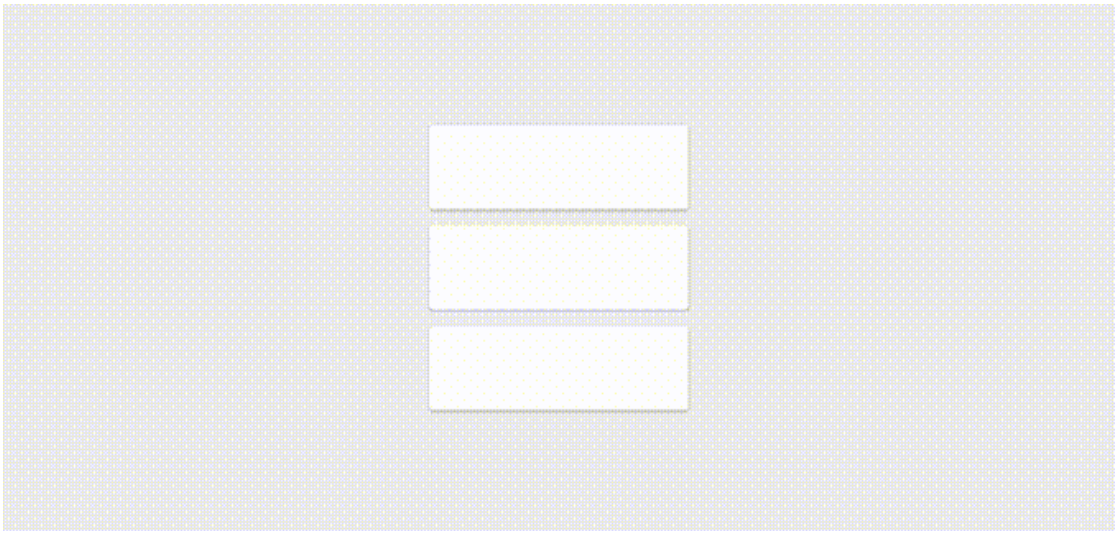


FIGURE 3.19. – L'axe Z est réservé à l'interaction utilisateur. Crédit : Google Design

3.3. Les nouveaux composants graphiques

3.3.1. Le menu latéral

Le menu latéral est un composant lié à la navigation. Il se caractérise par un panneau latéral qui se déplace depuis l'extérieur de l'écran jusqu'à un certain niveau à l'intérieur. En général, il se place sur la gauche de l'écran mais il est possible de le placer à droite, voire d'en placer deux, un à gauche et un à droite de l'écran.

Ce type de navigation est utilisé pour des grosses applications avec des parties distinctes. Il serait tout à fait légitime de l'utiliser pour des applications mobiles pour des sites comme Zeste de Savoir. C'est d'ailleurs le type de navigation utilisé sur la version mobile du site responsive. Il affiche l'utilisateur connecté, des liens vers les trois grandes catégories du site à savoir les forums, les articles et les tutoriels et un menu contextuel en fonction de l'emplacement de l'utilisateur.

FIGURE 3.20. – Exemple d'un menu latéral. Crédit : Google Blog Developer

3.3.2. Les composants flottants

3.3.2.1. Les labels flottants

Lorsque vous concevez un formulaire, vous avez besoin de un ou plusieurs `EditText`. Ce composant permet à l'utilisateur de saisir du texte dans un espace dédié

1. "Materiel" dans Materiel Design n'est pas anodin. Tous les éléments graphiques qui composent ce design est vu comme un matériel. C'est pourquoi nous utiliserons ce terme tout au long du tutoriel.
2. "Plan" sera utilisé pour qualifier la zone complète pour concevoir nos interfaces et avec lequel nous pouvons élever nos matériels pour afficher une ombre portée.
3. Si vous êtes déjà un développeur Android, cette unité ne devrait pas vous posez des problèmes. Pour les autres, "dp" signifie "Density-independent pixels" ce qui se traduit par une unité abstrait basé sur la densité physique de votre écran. Cela permet notamment de pouvoir utiliser le même nombre de dp pour des écrans qui se différencie peu dans leurs tailles.

à cet effet. Par contre, donnez du sens à ce composant était fastidieux. Vous deviez créer un `TextView` au dessus pour lui donner un titre et y placez un `hint`[4].

Avec la bibliothèque de design, vous pouvez spécifier un label flottant pour votre champ. Ce label se place dans le champ lorsque le curseur n'est pas dessus et se déplace en dehors lorsque vous y placez le curseur.

FIGURE 3.21. – Le label d'un EditText bouge quand le curseur est placé dessus. Crédit : Google

3.3.2.2. Les boutons flottants

Le bouton flottant est rapidement devenu le composant le plus répandu dans le Material Design. Tellement que certains développeurs pensent que le Material Design se limite à ce composant et aux cartes de Google Now. Or, même si ce bouton représente beaucoup dans le Material Design, il n'en reste qu'un composant parmi tous les autres.

FIGURE 3.22. – Le bouton standard du Material Design. Crédit : Google Blog Developer

3.3.3. Snackbar

La Snackbar vient remplacer les Toast sur Android. Pour rappel, les Toast étaient des petits messages avec un fond noir. Ils apparaissaient en bas des applications avec des bords arrondis sur les versions 4 d'Android et des bords carrés sur les version précédentes. Ils étaient utilisés pour afficher des messages courts suite à une action de l'utilisateur.

La Snackbar a exactement le même objectif. Afficher du texte, voire proposer une action liée à son message. Elle vient remplacer les Toast suite à une demande des utilisateurs Android qui s'attachaient à des alternatives aux Toasts, notamment à des croutons qui affichaient des messages avec la même animation que la Snackbar mais en haut de l'application.

FIGURE 3.23. – Apparition d'une Snackbar dans une application. Crédit : Google Blog Developer

3.3.4. Onglets

Visuellement, les onglets sont proches des anciens onglets. Pour cause, c'est techniquement le même composant mais avec un visuel adapté à la nouvelle charte du système. Le changement se situe dans la gestion technique de ce composant avec la venue de `TabLayout`. Ce nouveau layout simplifie grandement la gestion des onglets fixes, des onglets scrollables horizontalement et la liaison avec des composants du type `ViewPager`.

FIGURE 3.24. – Affichage des onglets Material Design. Crédit : Google Blog Developer

3.3.5. Extension de la Toolbar

La `Toolbar` est déjà une nouveauté. Avec le Material Design, vous êtes enfin complètement maitre de l'affichage de la barre d'action en spécifiant ce nouveau composant comme remplaçant à la barre d'action.

Mais les nouveautés ne s'arrêtent pas là puisque la `Toolbar` offre une extension pour un rendu plus visuel et attrayant. Vous pouvez simplement prolonger la couleur de la barre d'action comme le montre l'exemple de cette section ou placer des images. Pour vous donnez un exemple, c'est un composant qui convient parfaitement à des lecteurs audio pour afficher la pochette de l'album.

Notez le bouton à gauche de l'extension qui disparait lorsque vous refermez l'extension de la `Toolbar`. Il est à la fois raccord avec les bonnes pratiques de destruction des matériels mais s'affiche à gauche dans un espace dédié aux images pour garder une cohérence dans l'interface. Nous en parlerons peu dans ce tutoriel mais sachez que cet espace est souvent utilisé dans les applications Material Design, voire vivement encouragé sans pour autant être mentionné dans la documentation des bonnes pratiques.

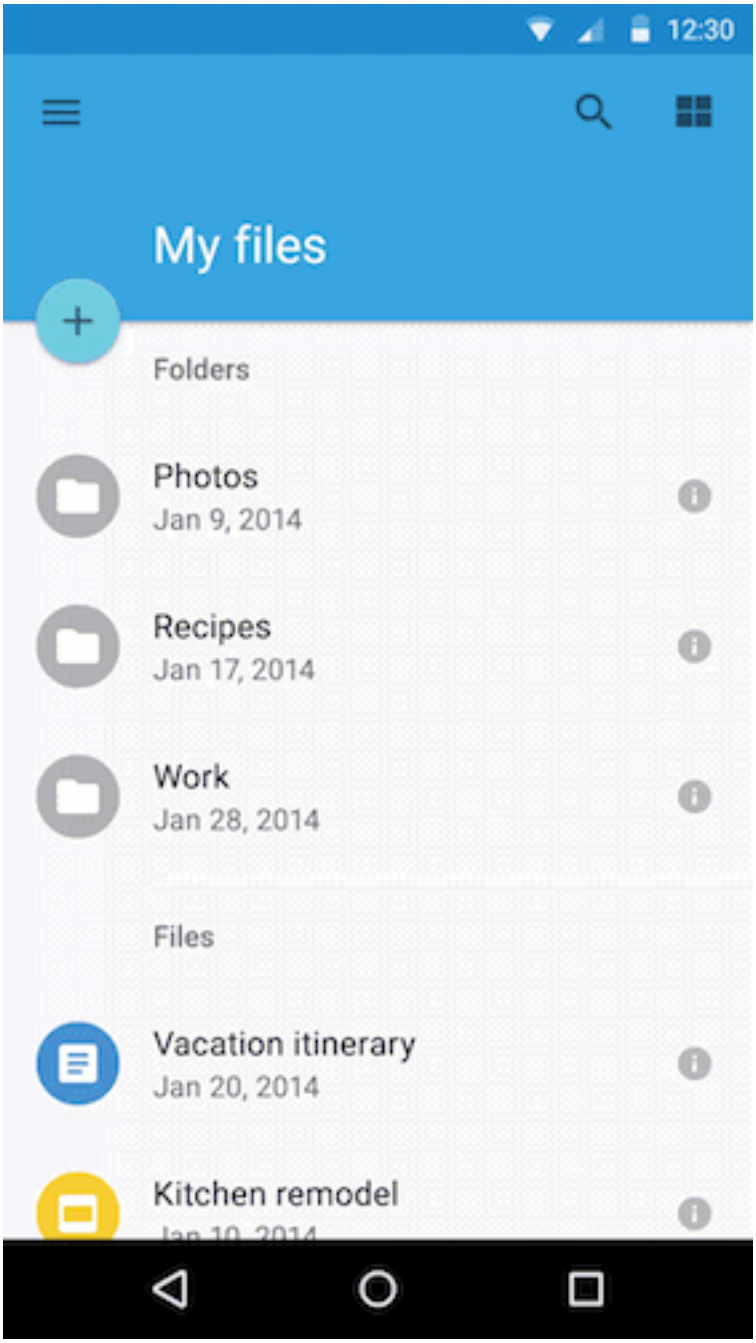


FIGURE 3.25. – Rendu de l'extension de la Toolbar. Crédit : Google Blog Developer

4. Le `hint` se représentait par du texte affiché dans le champ mais qui disparaissait lorsque le curseur était placé dessus.

4. La Toolbar, le composant Material Design pour la barre d'action

Si la `Toolbar` (doc) est un composant que vous ne connaissez pas, cette partie du tutoriel a de grandes chances de vous intéresser. Que vous soyez un débutant complet ou un familier des anciennes pratiques notamment avec l'usage de l'`ActionBar` (doc), lisez cette partie qui vous permettra de vous mettre à jour dans la gestion de la barre d'action de vos applications.

La section 1 se consacre à une explication du rôle de la `Toolbar` dans le Material Design : pourquoi est-ce que Google a jugé nécessaire de changer de composant et quelles sont les différences entre l'ancienne et la nouvelle barre d'action ? La section 2 explique comment intégrer basiquement une `Toolbar`, migrer de l'`ActionBar` vers la `Toolbar` et appliquer un style basique. Pour finir, la section 3 aborde les personnalisations communes que vous rencontrez régulièrement dans les autres applications Android.

4.1. A chaque situation, sa navigation

La navigation d'une application mobile peut devenir une tâche complexe lorsqu'elle doit afficher un grand nombre d'écrans avec des informations différentes. C'est pourquoi Android propose plusieurs composants visuels, chacun étant adapté à un type de navigation.

Tous ces composants sont utilisés au bon vouloir des développeurs. Il existe quand même des bonnes pratiques établies par Google pour garantir la meilleure expérience possible pour les utilisateurs finaux. Ce sont ces pratiques qui seront expliquées dans le cadre de ce tutoriel.

Parmi les navigations, nous retrouvons le *screen slide*. Il est utilisé pour mettre en place un enchainement d'écran par des onglets situés sous la barre d'action. Cette même barre d'action peut être utilisée pour la navigation. Par exemple, pour afficher les commentaires d'un article. Quant au menu latéral, il est utilisé

pour les applications volumineuses où il y a besoin de distinguer plusieurs parties distinctes.

Si nous prenons l'exemple d'un client mail, un menu latéral est presque obligatoire pour une expérience utilisateur optimale. Cela serait anti ergonomique de naviguer à travers tous les écrans par un *screen slide*, par la barre d'action ou par d'autres moyens de navigation personnalisés ou non.

Si nous prenons un dernier exemple, une application de messagerie possède rarement un menu latéral. Elle se contente d'avoir une liste de conversations et un fil de discussion quand l'utilisateur sélectionne une conversation. Le menu latéral ne serait pas adapté pour afficher cette liste de conversations.

4.2. Rôle de la Toolbar au sein du Material Design

Avant le Material Design, la barre d'action était un composant fixe, difficilement personnalisable et nommée par une `ActionBar`. Ce composant est fortement lié à une `Activity` et totalement géré par le framework Android. Lollipop vient changer la donne et propose un nouveau composant. La `Toolbar` est une généralisation des barres d'action utilisée dans les *layout* des applications. Ce nouveau composant peut être placé n'importe où dans une interface au bon vouloir du développeur. Il en revient à lui de désigner une `Toolbar` comme la barre d'action d'une `Activity`.

Le Material Design amène de nouveaux besoins et change les réflexions quant à l'élaboration de vos interfaces. L'ajout de souplesse dans la `Toolbar` permet de coller au mieux à ces nouvelles contraintes :

- Titre : comportement déjà largement adopté depuis les débuts de la barre d'action, la possibilité de placer un titre (et un sous-titre optionnellement) en haut à gauche de la barre.
- Menu : placé en haut à droite de la barre d'action, les boutons destinés au menu de l'écran offre des actions sur le contenu affiché.
- Navigation : une barre d'action peut déclarer un bouton de retour, un bouton d'ouverture d'un menu latéral, un bouton de fermeture d'un écran ou d'autres boutons plus personnels aux applications. Contrairement au menu, ces boutons devraient toujours être utilisés pour naviguer à travers des écrans.
- Etiqueter des images couvertures : placer une image en couverture dans la barre d'action attire l'attention de l'utilisateur et lui permet d'attacher un visuel au contenu qu'il est en train de consulter.

— Des vues personnalisées : la `Toolbar` n'est qu'un contenant comme les autres. Les développeurs ont le loisir d'y placer leurs vues personnalisées qui respectent leur charte graphique.

Toutes ces responsabilités peuvent déstabiliser. A quel moment est-il plus judicieux d'étendre la barre d'action pour y placer une image ou dois-je développer une barre d'action avec un titre, des menus, des boutons de navigation, l'étendre et y placer une ou deux vues personnalisées ?

Bien entendu, il existe des bonnes et des mauvaises pratiques dont voici les commandements.

1. **Tu nommeras tes écrans avec un titre.** Pour ne pas perdre votre utilisateur, c'est important d'afficher un titre à la barre d'action de vos applications et qui représente bien la fonctionnalité principale de l'écran courant. Il n'est pas interdit d'utiliser du jargon spécifique qui parle que à vos utilisateurs mais tentez d'être le plus clair et compréhensible possible.
2. **Tu ne cumuleras pas les barres en haut de ton écran.** Comme vous pouvez spécifier autant de barres que vous voulez dans vos fichiers XML, vous pouvez techniquement en rajouter plusieurs, à des endroits différents et avec des styles spécifiques. C'est une mauvaise idée. Ne placez qu'une seule barre et trouvez d'autres composants natifs pour les actions que vous avez placé dans les autres barres. Les utilisateurs Android ont l'habitude d'avoir qu'une seule barre, il n'y a pas de raison que votre application déroge à cette règle.
3. **Tu économiseras de la place.** Cette règle s'applique de moins en moins de nos jours avec les écrans qui grandissent de plus en plus mais il existe toujours des appareils avec des petits écrans. Pour ceux là, pensez qu'il faut exposer le plus possible le contenu de votre écran. Par exemple, prenons un écran qui affiche une liste et une barre d'action. Un comportement souhaité est de masquer la barre d'action dès que l'utilisateur commence à descendre dans la liste. Il n'est plus utile d'afficher la barre d'action si l'utilisateur est occupé avec autre chose. Pensez simplement à re-afficher la barre quand l'utilisateur remonte dans la liste. C'est le comportement habituel des barres avec des listes. Sachez que cette règle redevient importante depuis qu'un utilisateur peut afficher 2 applications en même temps.
4. **Tu utiliseras judicieusement une couverture.** Dès que votre écran peut être illustré par une image, il est recommandé d'étendre votre barre et de placer une image qui prend toute la place. Dès que l'utilisateur commence

à naviguer sur l'écran, réduisez cette barre et retirez peu à peu le visuel de cette image. Attention, des pièges peuvent vite arriver. Si vous disposez d'image en basse qualité ou pas assez grande, évitez de l'utiliser comme couverture. Le rendu sera médiocre et vos utilisateurs mécontents. Assurez vous que toutes vos images seront d'une qualité suffisante.

5. **Tu n'abuseras pas des composants de navigation.** Une barre d'action peut être rendu compatible avec un menu latéral et définir un bouton de retour. Cette combinaison est un exemple typique de composants incompatibles. Si vous rendez compatible votre barre d'action avec le menu latéral, le bouton tout à gauche de la barre est utilisé pour ouvrir ce menu. Mais le bouton de retour se place exactement à la même place. Ce bouton ne pouvant pas avoir deux rôles, vous devez choisir le composant le plus approprié pour votre écran.
6. **Tu garderas une cohérence.** Les écrans peuvent avoir des barres d'action avec des titres, des menus ou des navigations différentes mais dans cette diversité, c'est important de garder une certaine cohérence. Par exemple, une application avec un menu latéral qui rend accessible les grandes fonctionnalités de l'application et des sous-écrans qui sont plus spécifiques. Un comportement usuel est une barre d'action compatible avec le menu latéral pour les écrans des grandes fonctionnalités et un bouton de retour à la place pour tous les autres écrans plus spécifique.
7. **Tu feras preuve de bon gout dans tes styles.** Une application possède une charte graphique qui se conserve à travers tous les écrans. Vous êtes libre de mettre au point la charte que vous souhaitez mais tentez de la respecter.

4.3. Ajouter une Toolbar à votre application

Disons-le, le prix à payer pour une plus grande maitrise de la barre d'action se fait au détriment d'une simplicité d'usage pour le développeur mais apporte une plus grande personnalisation et flexibilité. Auparavant, la barre d'action était automatiquement rajoutée pour les terminaux sous Android 3 et supérieur. La migration va demander du travail, parfois longue si votre application existante est complexe dans son interface.

4.3.1. Utiliser une Toolbar comme une ActionBar

Commencez par spécifier les dépendances nécessaires dans le fichier `build.gradle` de votre application ou module :

[Dépendances nécessaires pour la Toolbar et son design]groovy compile 'com.android.support :appcompat-v7 :24.2.0' compile 'com.android.support :support-v4 :24.2.0' compile 'com.android.support :design :24.2.0'

Vous remarquerez que les versions des bibliothèques sont identiques. Sachez que Google tient à garder le plus possible cette cohérence entre ses bibliothèque pour ne pas perdre le développeur entre tous les numéros de version. Vous ne risquez donc pas grand chose à spécifier le même numéro pour toutes ces bibliothèques et n'hésitez pas à faire confiance à Android Studio quand il vous signale qu'il y a une nouvelle version plus à jour que celle que vous avez spécifié. La version 24.2.0 spécifiée dans ce tutoriel ne correspondra pas éternellement à la dernière version en date. Ne vous étonnez donc pas d'avoir un numéro de version différent du mien !

Une fois que vous avez accès à toutes les ressources indispensables à la déclaration d'une `Toolbar`, la première chose à faire est la création ou modification du style de votre application. Votre style doit obligatoirement hériter du thème parent `Theme.AppCompat.*`. La suite de ce style parent dépend alors de ce que vous désirez. Vous avez le choix entre un thème dit *light* ou *dark*, et pour ces deux thèmes, avec automatiquement une `Toolbar` ou non.

Choisir le thème avec une `Toolbar` me permet-il pas de faire une migration simple et rapide depuis l'ancien système de la barre d'action ? Oui, tout à fait mais il y a plusieurs choses à savoir.

1. Si vous utilisez qu'un seul style pour toute votre application mais que votre application comporte des écrans sans barre d'action, vous devrez spécifier un style particulier pour ces écrans. Typiquement, le même que le style général mais dans son mode sans `Toolbar`.
2. Vous n'avez aucune maitrise sur la hiérarchie des vues autour de la barre d'action. Nous verrons plus loin dans ce chapitre que cette `Toolbar` peut vite montrer ses limites.

En conclusion, utilisez la `Toolbar` automatique que si vous êtes certain de n'avoir jamais besoin d'étendre la barre d'action ou d'y ajouter des comportements spéciaux.

Vous l'aurez compris, dans le cadre de ce tutoriel, nous allons déclarer un style qui ne place aucune barre d'action automatiquement pour des raisons évidentes de

pédagogie.

[Style Material Design issu de la bibliothèque de compatibilité]xml <!– Base application theme. –> <style name=”AppTheme” parent=”Base.Theme.ZdS”> </style>

<style name=”Base.Theme.ZdS” parent=”Theme.AppCompat.Light.NoActionBar”> <item name=”colorPrimary”>@color/primary</item> <item name=”colorPrimaryDark”>@ $/item >< itemname = "colorAccent" > @color/accent < /item >< itemname = "android : textColorPrimary" > @color/primary_text < /item >< itemname = "android : windowBackground" > @color/window_background < /item >< /style >$

Expliquons un peu les personnalisations placées dans notre style qui ont été introduites depuis la venue du Materiel Design pour la plupart.

— `colorPrimary`, `colorPrimaryDark` et `colorAccent` définissent les couleurs principales de votre application. Votre barre d'action prendra la couleur de `colorPrimary`, la barre d'état prendra la couleur de `colorPrimaryDark` et toutes les actions secondaires (comme les boutons flottants) prendront la couleur de `colorAccent`. Pensez à choisir des couleurs harmonieuses pour ces 3 propriétés.
— `android:textColorPrimary` définit la couleur de votre texte **en dehors de la barre d'action** et `android:windowBackground` définit la couleur du fond de votre application. Si vous choissiez un fond clair, pensez à une couleur de texte foncé et inversément si vous êtes sur un fond foncé.

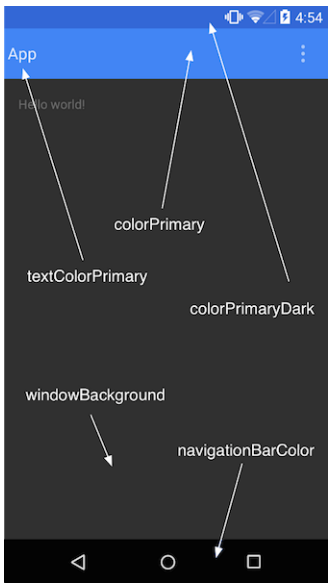


FIGURE 4.1. – Personnalisation d'un thème Material Design.

Une propriété préfixée par `android:` est une propriété qui existait bien avant le Material Design et disponible directement dans le framework Android. S'il n'y a aucun préfixe, cela veut dire que le style a été déclaré en dehors du framework Android. Dans notre cas précent, ces propriétés ont été déclarées dans la bibliothèque de compatibilité sur le design.

La `Toolbar` est une vue comme les autres. Si votre thème dit explicitement que vous ne voulez pas automatiquement ce composant sur vos écrans, comme c'est le cas pour nous, vous devez la placer vous même dans les *layout* de vos écrans. L'exemple le plus basique est le suivant.

[Exemple basique du placement d'une Toolbar]xml < ?xml version=”1.0” encoding=”utf-8” ?> <LinearLayout xmlns :android=”http ://schemas.android.com/apk/res/android” xmlns :app=”http ://schemas.android.com/apk/res-auto” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$: $orientation$ = "$vertical$" >

<android.support.v7.widget.Toolbar android :id=”@+id/toolbar” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "?$attr/actionBarSize$"$android$: $background$ = "?$attr/colorPrimary$"/ >

<FrameLayout android :id=”@+id/content” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"/ >

</LinearLayout>

Nous plaçons en haut de l'écran le composant, nous lui spécifions la hauteur standard du composant grâce à la valeur `?attr/actionBarSize` et nous lui donnons la couleur primaire, `?attr/colorPrimary`, que nous avons précédemment configuré dans notre style. Mais même si visuellement, vous disposez bien d'une barre d'action en haut de votre écran, elle n'est pas encore vue comme telle. Pour qu'elle soit considérée comme une ActionBar, vous devez le dire explicitement à votre `Activity` qui doit obligatoirement hériter de `AppCompatActivity`.

[Renseigne la Toolbar comme l'ActionBar de l'écran]java public class ToolbarActivity extends AppCompatActivity @Override protected void onCreate(@Nullable Bundle savedInstanceState) super.onCreate(savedInstanceState) ; setContentView(R.layout.activity$_{t}o$

Vous disposez dès lors d'un écran avec une `Toolbar` considérée comme une `ActionBar`. Vous pouvez y ajouter tous les comportement supplémentaires que vous voulez et que nous allons abordé dans la suite de ce chapitre.

4.3.2. Appliquer un style

Le style général que nous avons appliqué à notre application est un premier pas vers la personnalisation de la barre d'action mais nous pouvons aller plus loin. Nous pouvons personnaliser le titre, la couleur du texte des menus, la couleur des retours utilisateurs sur les boutons du menu et le popup des sous-menus. Reprenons notre `Toolbar` et appliquons lui 3 nouvelles propriétés.

[Applique un style spécifique à une Toolbar]xml <android.support.v7.widget.Toolbar android :id=”@+id/toolbar” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "?$attr/actionBarSize$"$android$: $background$ = "?$attr/colorPrimary$"app : $popupTheme$ = "@$style/ThemeOverlay.AppCompat.Light$"$app$: $theme$ = "@$style/ThemeOverlay$. $titleTextAppearance$ = "@$style/TextAppearance.Randoomz.Toolbar.Title$"/ >

- `app:popupTheme` et `app:theme` applique, respectivement, le style pour le popup des sous menus et pour la `Toolbar`. Ces deux propriétés doivent renseigner comme thème `ThemeOverlay.AppCompat.*` ou un style personnel qui hérite de ce thème.
- `app:titleTextAppearance` applique le style pour le titre de la barre d'action. Cette propriété doit renseigner comme thème `TextAppearance.Widget.AppCompat.Toolbar.Title` ou un style personne qui hérite de ce thème.

Dans notre exemple, nous avons créer des thèmes personnels pour `app:theme` et `app:titleTextAppearance`, sur lesquelles nous allons rapidement revenir. Quant à `app:popupTheme`, nous avons spécifié le thème `ThemeOverlay.AppCompat.Light`. Cela permettra d'avoir des *popup* avec un fond clair, plus précisément blanc cassé.

Maintenant, regardons nos thèmes personnels dans nos styles.

[Style utilisé pour la Toolbar]xml <style name=”TextAppearance.Randoomz.Toolbar.Title” parent=”TextAppearance.Widget.AppCompat.Toolbar.Title”> <item name=”android :textSize”>21sp</item> <item name=”android :textStyle”>italic</item> </style>

<style name=”ThemeOverlay.Randoomz.Toolbar” parent=”@style/ThemeOverlay.AppCompat.Dark.ActionBar”> <item name=”android :textColorPrimary”>@android :color/holo
$/item >< itemname$ = "$actionMenuTextColor$" > @$android$: $color/holo_{g}reen_{l}ight$ < $/item >< itemname$ = "$colorAccent$" > @$android$: $color/holo_{o}range_{d}ark$ < $/item >< itemname$ = "$colorControlNormal$" > @$android$: $color/holo_{p}urple$ < $/item >< itemname$ = "$colorControlActivated$" > @$android$: $color/holo_{r}ed_{d}ark$ <

$/item >< itemname = "colorControlHighlight" > @android : color/holo_red_dark < /item >< /style >$

Nous avons un premier thème qui est destiné à personnaliser le titre de notre barre d'action. Vous devrez aisément comprendre son contenu, nous spécifions simplement la taille et la mise en italique du titre. Le second thème est plus intéressant puisque c'est lui qui va personnaliser toutes les couleurs de la barre.

- `android:textColorPrimary` donne la couleur au titre de la barre.
- `actionMenuTextColor` donne la couleur des textes du menu affiché sur la barre.
- `colorAccent` donne la couleur des *checkboxes* que vous pouvez avoir dans vos menus.
- `colorControlNormal` donne la couleur de l'icône à 3 points destiné à afficher les menus cachés.
- `colorControlActivated` donne la couleur de certaines vues à l'état activé (comme les *switch* ou les *checkboxes*).
- `colorControlHighlight` donne la couleur des retours utilisateurs sur les boutons du menu.

Le résultat de ce style sur un écran avec un menu sera le suivant :

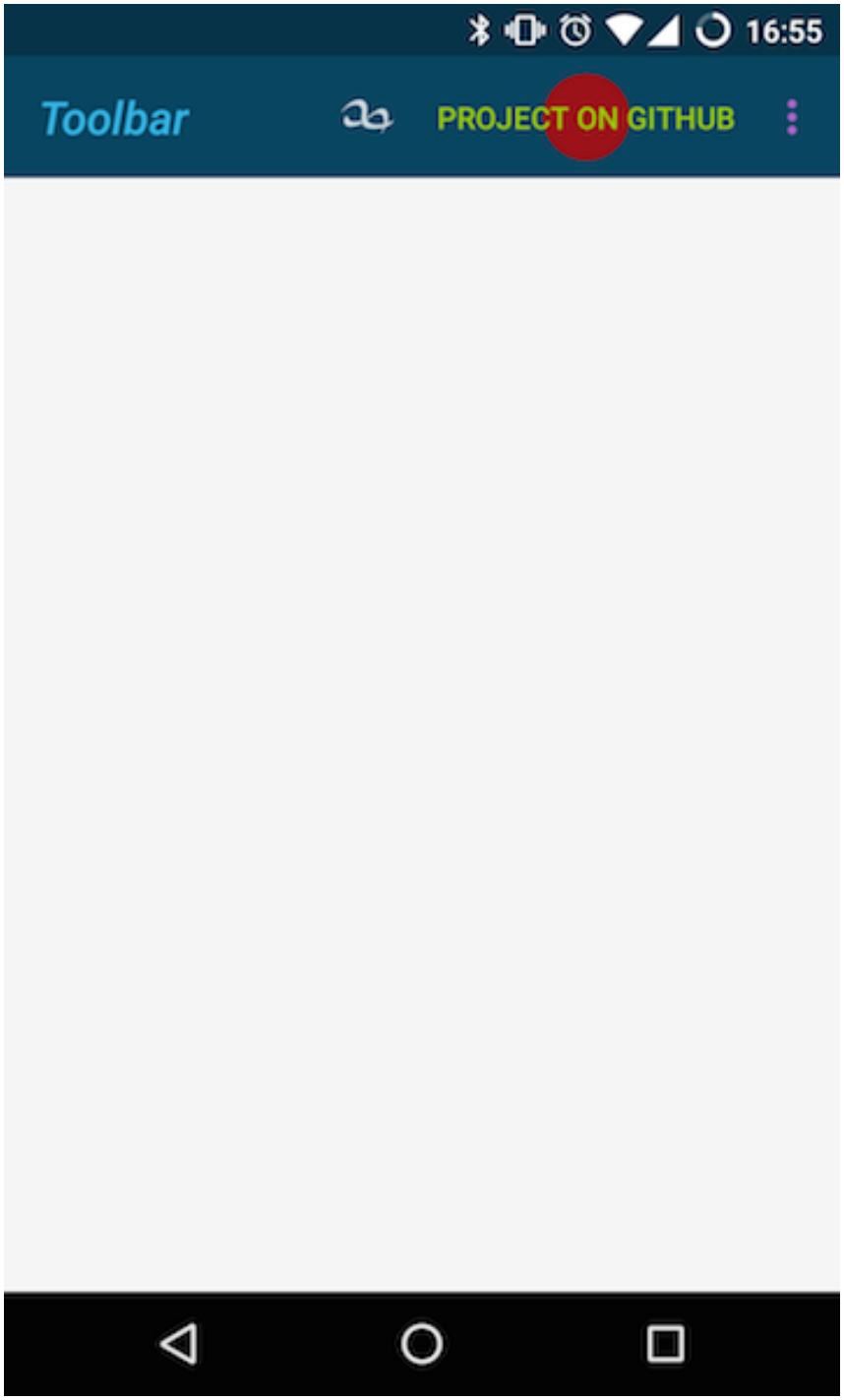


FIGURE 4.2. – Personnalisation d'une Toolbar

Mais bien entendu, vous ferez quelque chose de plus joli pour vos écrans.

4.4. Onglets attachés à la Toolbar

L'onglet est un composant de navigation répandu et utilisé depuis les toutes premières versions d'Android, mais son usage a évolué quasiment entre chaque version majeure d'Android. Aujourd'hui, vous allez découvrir son usage à travers la bibliothèque de compatibilité sur le design avec le composant `TabLayout` et couplé avec un `ViewPager`.

Un `TabLayout` peut déclarer directement dans son fichier XML des `TabItem` pour avoir des onglets fixes. Pourquoi n'allons-nous pas enseigner cette pratique dans ce tutoriel ? Simplement parce que l'usage le plus courant aujourd'hui est de rendre les onglets utilisables grâce à des *swipe* de l'utilisateur pour passer d'un onglet à l'autre et pour gérer plus beaucoup plus facilement toutes les interactions utilisateurs !

Partons du principe que notre application utilise un thème qui place automatiquement une `Toolbar` à notre écran (comme l'explique la section précédente), nous avons besoin que de placer un `TabLayout` et un `ViewPager` qu'il nous faudra lier dans l'`Activity` ou le `Fragment` hôte.

[Spécifier un TabLayout et un ViewPager]xml < ?xml version=”1.0” encoding=”utf-8” ?> <LinearLayout xmlns :android=”http ://schemas.android.com/apk/res/android” xmlns :app=”http ://schemas.android.com/apk/res-auto” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$: $orientation$ = "$vertical$" >

<android.support.design.widget.TabLayout android :id=”@+id/tabs” style=”@style/Widget.Randoomz.TabLayout” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$wrap_{c}ontent$"/ >

<android.support.v4.view.ViewPager android :id=”@+id/viewpager” android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "0px"$android$: $layout_{w}eight$ = "1"/ >< /$LinearLayout$ >

Notez qu'un style est placé sur le `TabLayout`. Il est en effet possible de personnaliser l'espace qui accueillera vos onglets et les onglets eux-même. Pour ce faire, il suffit de créer un style qui étend `Widget.Design.TabLayout` pour personnaliser l'espace des onglets et `TextAppearance.Design.Tab` pour les onglets eux-même.

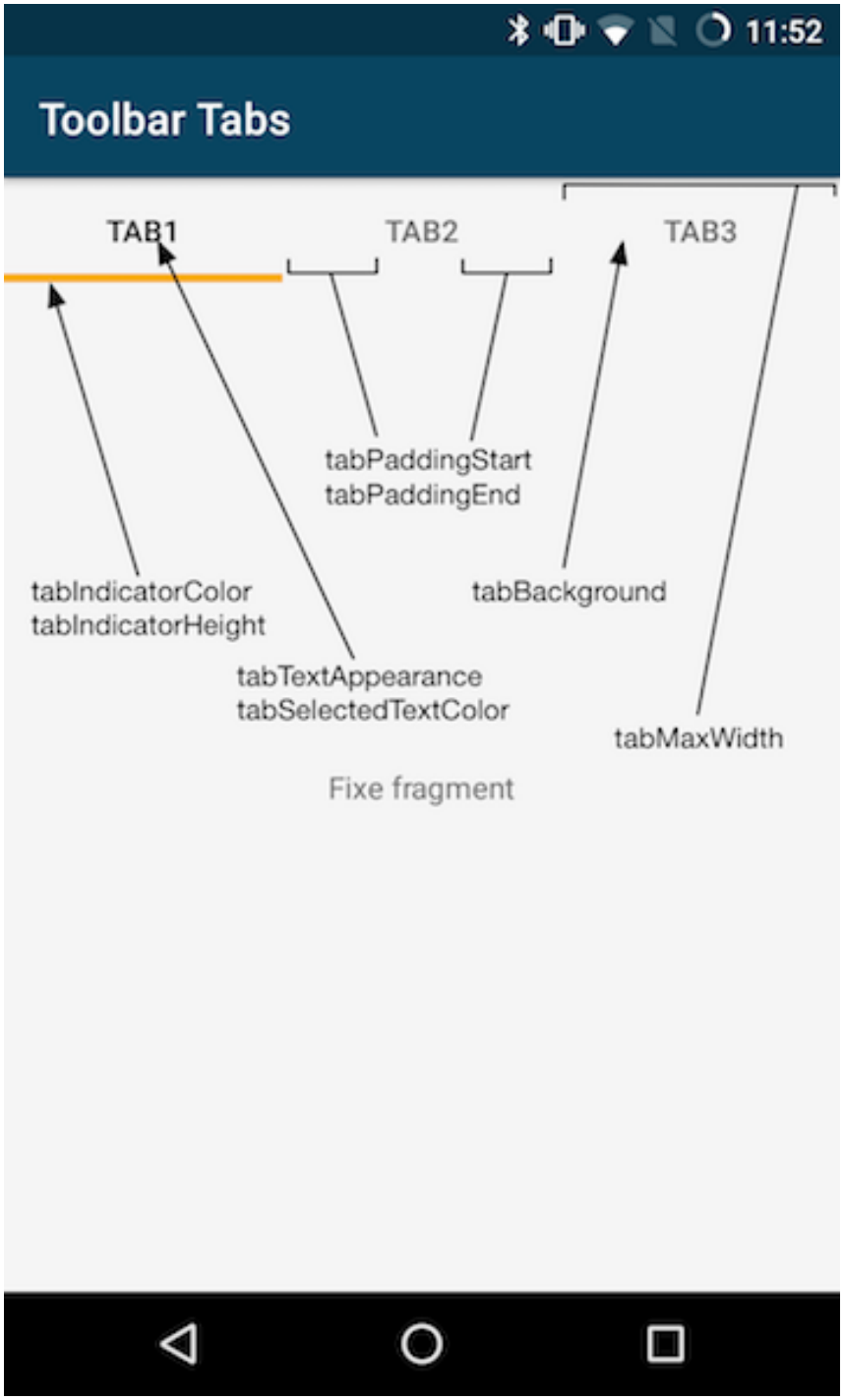
Voici un exemple de style que vous pourriez utiliser pour les onglets, `Tab` :

[Style d'un onglet]xml <style name=”TextAppearance.Randoomz.Tab” parent=”TextAppearance.Design.Tab”> <item name=”android :textSize”>14sp</item> <item name=”android :textColor”> ?android :textColorSecondary</item> <item name=”textAllCaps”>true</item> </style>

Dans cet exemple, nous personnalisons la taille, la couleur du texte des onglets et nous spécifions que toutes leurs lettres doivent être en majuscule. Plus intéressant maintenant, voici un exemple de style que vous pourriez utiliser sur un `TabLayout` et qui utilise notre précédent style sur les onglets.

[Style du conteneur des onglets]xml <style name=”Widget.Randoomz.TabLayout” parent=”Widget.Design.TabLayout”> <item name=”tabMaxWidth”>@dimen/tab$_{m}ax_{w}idth <$ $/item >< itemname = "tabIndicatorColor" >?attr/colorAccent < /item >< itemname = "tabIndicatorHeight" > 4dp < /item >< itemname = "tabPaddingStart" > 12dp < /item >< itemname = "tabPaddingEnd" > 12dp < /item >< itemname = "tabBackground" >?attr/selectableItemBackground < /item >< itemname = "tabTextAppearance" > @style/TextAppearance.Randoomz.Tab < /item >< itemname = "tabSelectedTextColor" >?android : textColorPrimary < /item >< /style >$

- `tabMaxWidth` donne la hauteur maximale de vos onglets.
- `tabIndicatorColor` donne la couleur à l'indicateur placé sous un onglet quand il est sélectionné.
- `tabIndicatorHeight` donne la hauteur de l'indicateur.
- `tabPaddingStart` place un espace à gauche de l'onglet pour éviter de coller son texte sur son bord.
- `tabPaddingEnd` place un espace à droite de l'onglet pour éviter de coller son texte sur son bord.
- `tabBackground` donne la couleur du fond des onglets. Gardez en tête que la couleur renseignée doit aussi gérer le retour visuel. A la place d'une simple couleur, préférez un `selector` pour renseigner les couleurs aux différents états de l'onglet.
- `tabTextAppearance` donne le style du texte des onglets. Nous renseignons ici notre précédent style.
- `tabSelectedTextColor` donne la couleur du texte de l'onglet quand il est sélectionné.



Vous avez là toutes les cartes en main pour personnaliser à fond vos onglets. Il ne nous reste plus qu'à lier le tout dans un écran en Java. Commençons par le `ViewPager`. Pour ceux qui ne connaissent pas ce composant, le `ViewPager`

permet de glisser d'une page à l'autre. Pour initialiser ce composant, il suffit de lui fournir un `FragmentPagerAdapter` qui aura la responsabilité de créer nos pages et nos onglets. Un adaptateur simple pourrait être la création de trois onglets, nommés "Tab1", "Tab2" et "Tab3", et qui va instancier un fragment pour chaque onglet.

[Adaptateur d'un ViewPager]java class PagerAdapter extends FragmentPagerAdapter private final String tabTitles[] = new String[]"Tab1", "Tab2", "Tab3" ; private final static int PAGE$_{C}OUNT = 3$;

PagerAdapter(FragmentManager fm) super(fm) ;

@Override public int getCount() return PAGE$_{C}OUNT$;

@Override public Fragment getItem(int position) return new FixeFragment() ;

@Override public CharSequence getPageTitle(int position) return tabTitles[position] ;

Maintenant que nous avons un adaptateur qui va s'occuper de créer nos pages et nos onglets, il nous suffit de l'attacher au `ViewPager` et de l'attacher lui-même à notre `TabLayout`. Le tout se faisait très simplement de la manière suivante :

[Lie le ViewPager au TabLayout]java @Override protected void onCreate(@Nullable Bundle savedInstanceState) super.onCreate(savedInstanceState) ; setContentView(R.layout.activity$_{t}o$

final ViewPager viewPager = (ViewPager) findViewById(R.id.viewpager) ; viewPager.setAdapter(new PagerAdapter(getSupportFragmentManager())) ;

final TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs) ; tabLayout.setupWithViewPager(viewPager) ;

Cette simplicité est déconcertante. Ces quelques lignes nous permettent de mettre en place facilement et rapidement un écran avec des onglets et des fragments indépendants pour chacun de ces onglets. Le voici suivant correspond à nos onglets avec le style précédemment expliqué.

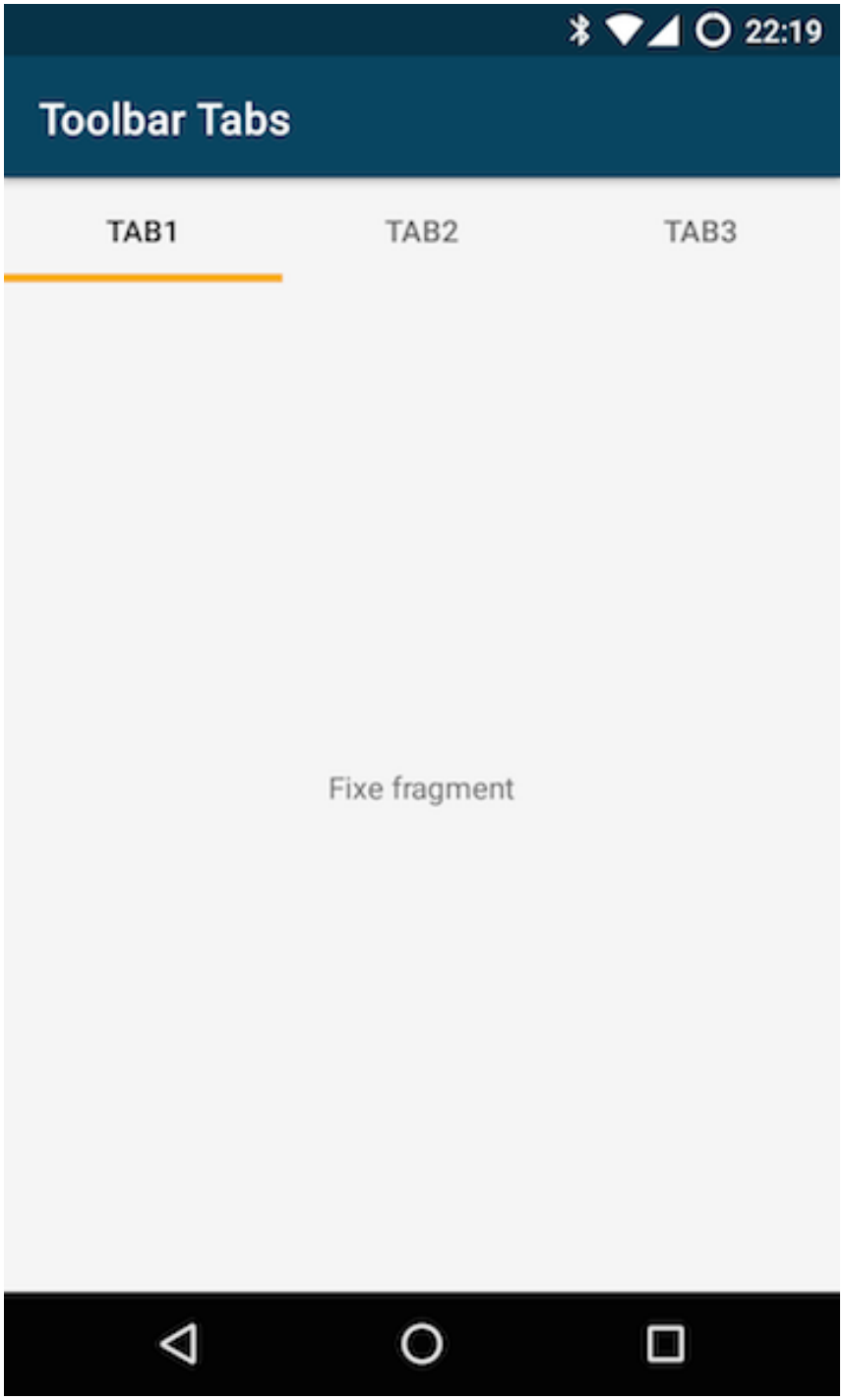


FIGURE 4.3. – Toolbar avec des onglets

4.5. Toolbar étendue et gestion du scroll

Une fois n'est pas coutume, nous allons d'abord voir ce que nous allons développer pour que vous comprenez bien le résultat attendue. Vous connaissez sans aucun doute cet effet. C'est le plus souvent utilisé pour des applications de musique ou de films, là où afficher une image dans la `Toolbar` prend tout son sens pour illustrer le contenu de l'écran. Par exemple, dans un écran qui affiche les détails d'un album de musique, la couverture de cet album pourrait figurer dans la `Toolbar`.

Nous, nous allons tenter de développer l'écran ci-dessous composé d'une `Toolbar`, d'une image de couverture facultative et d'une liste dans le composant `RecyclerView` qui va nous permettre de réagir avec la barre d'action.

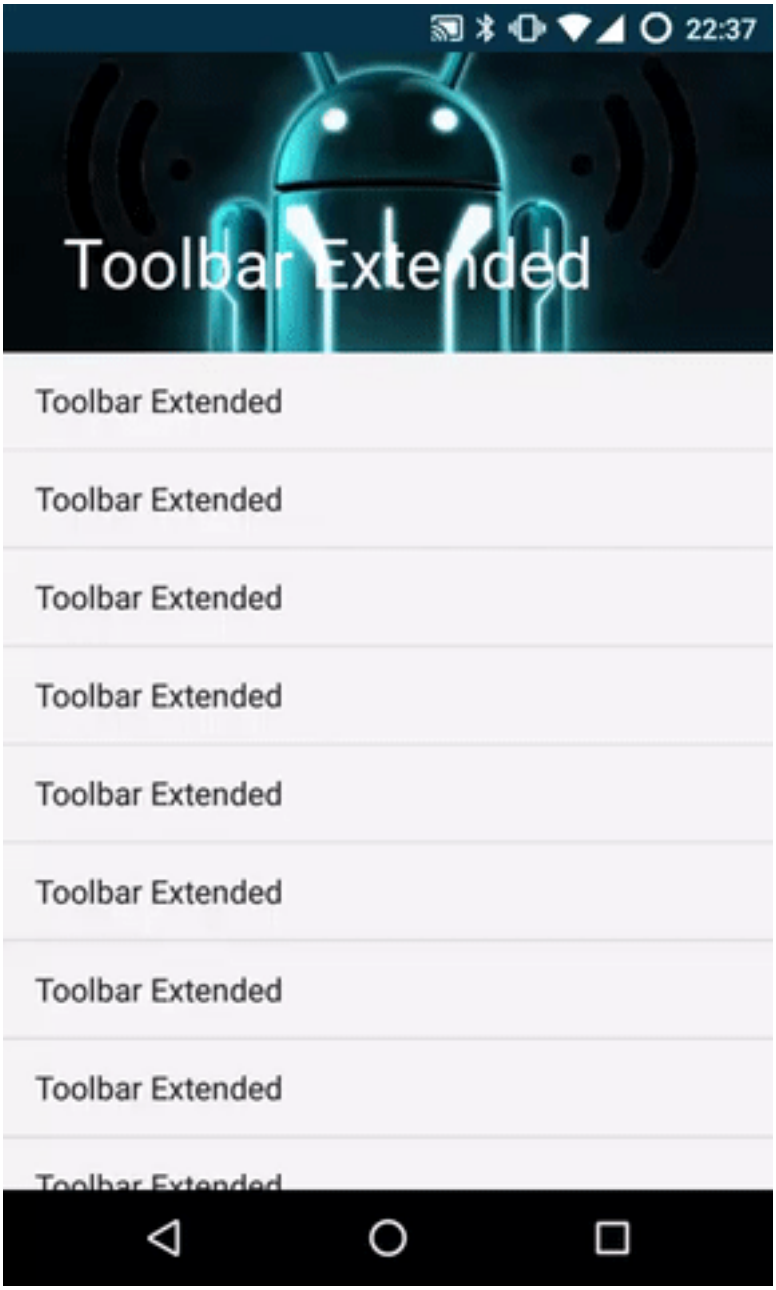


FIGURE 4.4. – Toolbar étendue avec une couverture

La première chose à faire consiste à renseigner un `CoordinatorLayout` comme conteneur racine à notre écran. Ce conteneur est une sous classe d'un `FrameLayout` et est utilisé pour spécifier les interactions avec ses vues filles. Dans notre cas présent, il va nous permettre de renseigner le comportement à adopter d'une liste par rapport à une `Toolbar`, que nous allons aussi devoir spécifier dans notre *layout*. Pour cette

liste, nous allons utiliser un `RecyclerView` mais libre à vous de choisir une autre vue du moment qu'elle est scrollable.

[Déclaration d'un CoordinatorLayout et d'une liste pour réagir avec la barre d'action]["16"]xml < ?xml version="1.0" encoding="utf-8" ?> <android.support.design.widget.CoordinatorLayout xmlns :android="http ://schemas.android.com/apk/res/android" xmlns :app="http ://schemas.android.com/apk/res-auto" android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$" >

< !– Toolbar here. –>

<android.support.v7.widget.RecyclerView android :id="@+id/list" android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$: $scrollbars$ = "$vertical$"app : $layoutManager$ = "$LinearLayoutManager$"app : $layout_{b}ehavior$ = "@$string/appbar_{s}crolling_{v}iew_{b}ehavior$"/ >

</android.support.design.widget.CoordinatorLayout>

Constatez l'attribut `app:layout_behavior` avec comme valeur `@string/appbar_scrolling_view_behavior`. Cette dernière ressource est détenue par la bibliothèque de compatibilité et permet de notifier la barre d'action des *scroll* sur la liste. Ainsi, la barre peut réagir de manière appropriée.

La seconde chose à faire consiste à renseigner un `AppBarLayout` comme conteneur parent de notre `Toolbar`. Ce conteneur est une sous classe d'un `LinearLayout` vertical. Il gère les mouvements de scrolling de la `Toolbar`. Ses enfants doivent fournir le comportement de scrolling désiré à travers l'attribut `app:layout_scrollFlags`. Cet attribut peut prendre les valeurs `scroll`, `enterAlways`, `enterAlwaysCollapsed`, `exitUntilCollapsed` et `snap` :

- `scroll` indique que nous voulons réagir au scroll.
- `enterAlways` permet d'afficher la barre d'action directement quand l'utilisateur *scroll* vers le haut. Normalement, le comportement par défaut est d'afficher la barre d'action dès que l'utilisateur se trouve en haut de la liste et non pas n'importe où dans la liste.
- `enterAlwaysCollapsed` permet d'afficher une hauteur supplémentaire à la `Toolbar` si elle est renseignée (pour afficher une couverture par exemple). Cette valeur va vous permettre d'étendre jusqu'au bout la barre d'action avant de continuer le *scroll*.
- `exitUntilCollapsed` permet de faire disparaitre entièrement la barre d'action quand l'utilisateur *scroll* dans la liste.
- `snap` permet d'avoir un petit effet de rebond quand l'utilisateur *scroll* qu'un petit peu dans la liste. Cet effet est montré sur notre écran présenté en début de section.

Je vous laisse à votre bon soin pour faire des tests en appliquant ces valeurs sur votre `Toolbar` mais nous, nous voulons en plus étendre la barre d'action et y placer une couverture. Pour parvenir à cet effet, nous devons placer un `CollapsingToolbarLayout` entre notre `AppBarLayout` et notre `Toolbar`. Ce conteneur est une sous classe d'un `FrameLayout` et gère tous les attributs de la barre étendue.

- `app:contentScrim` est la couleur affiché quand la barre d'action atteint une certaine hauteur.
- `app:expandedTitleMarginEnd` et `app:expandedTitleMarginStart` sont les marges à gauche et à droite du titre de la barre d'action quand elle est étendue.
- `app:layout_collapseMode` se place sur les vues filles du conteneur et peut prendre comme valeur `pin` ou `parallax`. Ils signifient respectivement que la barre d'action doit être fixé en haut de l'écran et qu'il doit y avoir un effet de parallaxe sur la vue.

Pour finir, si vous désirez placer une couverture, il suffit de mettre une `ImageView` au même niveau que la `Toolbar`. La totalité du *layout* qui correspond à notre écran voulu correspond donc au code XML suivant.

[Layout utilisé pour une barre d'action étendue et une couverture]xml < ?xml version=”1.0” encoding=”utf-8” ?> <android.support.design.widget.CoordinatorLayout xmlns :android=”http ://schemas.android.com/apk/res/android” xmlns :app=”http ://schemas.android.com/apk/res-auto” android :layout$_w idth$ = "$match_p arent$"$android$: $layout_h eight$ = "$match_p arent$" >

<android.support.design.widget.AppBarLayout android :layout$_w idth$ = "$match_p arent$"$android$: $layout_h eight$ = "150dp"$android$: $theme$ = "@$style/ThemeOverlay.AppCompat.Dark.ActionBar$" >

<android.support.design.widget.CollapsingToolbarLayout android :id=”@+id/collapsing$_t oolbar$"$andr$ $layout_w idth$ = "$match_p arent$"$android$: $layout_h eight$ = "$match_p arent$"app : $expandedTitleMarginE$ "64dp"app : $expandedTitleMarginStart$ = "30dp"app : $layout_s crollFlags$ = "$scroll|exitUntilCollap$

<android.support.v7.widget.Toolbar android :id=”@+id/toolbar” android :layout$_w idth$ = "$match_p arent$"$android$: $layout_h eight$ = "?$attr/actionBarSize$"app : $layout_s crollFlags$ = "$scroll|enterAlways$"/ >

< !– Optional : Used to display a cover. –> <ImageView android :layout$_w idth$ = "$match_p arent$"$android$: $layout_h eight$ = "$match_p arent$"$android$: $scaleType$ = "$centerCrop$"$android$: src = "@$drawable/android_l andscape$"$app$: $layout_c ollapseMode$ = "$parallax$"app : $layout_s crollFlags$ = "$scroll|enterAlways$"/ >< /$android.support.design.widget.Co$ /$android.support.design.widget.AppBarLayout$ >

<android.support.v7.widget.RecyclerView android :id="@+id/list" android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$: $scrollbars$ = "$vertical$"app : $layoutManager$ = "$LinearLayoutManager$"app : $layout_{b}ehavior$ = "@$string/appbar_{s}crolling_{v}iew_{b}ehavior$"/ >< /$android.support.design.widget.CoordinatorLayout$ >

4.6. Recherche

La recherche est accessible par un bouton du menu qui va s'étendre sur toute la largeur de la barre quand l'utilisateur cliquera dessus. Après quoi, lors de la saisie d'une recherche, nous pourrons écouter chaque caractère rentré ou écouter la soumission de sa recherche. C'est au choix et nous allons apprendre à gérer ces 2 options.

Dans un premier temps, nous allons devoir créer un menu pour notre recherche. Renseignez l'icône qui vous semble la plus appropriée, la valeur `collapseActionView` dans l'attribut `app:showAsAction` pour que le menu réagisse à son ouverture sur tout le long de la barre et enfin la valeur `android.support.v7.widget.SearchView` dans l'attribut `app:actionViewClass` pour spécifier la vue standard de la recherche.

[Menu pour la recherche]xml < ?xml version="1.0" encoding="utf-8" ?> <menu xmlns :android="http ://schemas.android.com/apk/res/android" xmlns :app="http ://schemas.android.com/apk/res-auto" xmlns :tools="http ://schemas.android.com/tools" tools :context="org.randoomz.zdsmessenger.ui.search.SearchActivity"> <item android :id="@+id/ac $icon$ = "@$drawable/ic_{s}earch$"$android$: $title$ = "@$string/menu_{s}earch$"app : $actionViewClass$ = "$android.support.v7.widget.SearchView$"$app$: $showAsAction$ = "$ifRoom|collapseActionView$"/ > /$menu$ >

Dans un second temps, nous allons renseigner la configuration de notre écran de recherche grâce à un fichier xml qui doit être rangé dans le dossier `res/xml`. Ce fichier xml n'est pas spécifique au Material Design, ni à la `Toolbar`. Il n'a pas changé depuis les débuts d'Android et renseigne le message qui invite à la recherche dans le composant et des modes complémentaires si vous le désirez. Pour ce dernier cas, vous pouvez demander au framework Android de rajouter automatiquement la reconnaissance de voix grâce aux valeurs `showVoiceSearchButton` et `launchRecognizer`. C'est ce que nous renseignons dans notre recherche.

[Configuration de la recherche]xml < ?xml version="1.0" encoding="utf-8" ?> <searchable xmlns :android="http ://schemas.android.com/apk/res/android" android :hint="@string $label$ = "@$string/app_{n}ame$"$android$: $voiceSearchMode$ = "$showVoiceSearchButton|launchRecogn$

Dans un troisième temps, nous devons lier ce dernier fichier de configuration à notre `Activity` qui contient notre écran de recherche. De plus, si nous le désirons, nous pouvons informer le système que notre application contient un écran de recherche. Cela permettra à d'autres applications de lancer notre `Activity` pour effectuer la recherche souhaitée. Si nous partons de l'hypothèse que notre `Activity` se nomme `ToolbarSearchActivity`, nous renseignerons la déclaration suivante de notre écran dans notre *manifest*.

[Déclaration de notre Activity de recherche dans notre fichier manifest]xml <activity android :name=”org.randoomz.demo.design.toolbar.ToolbarSearchActivity” android :label=”@string/title$_{d}esign_{t}oolbar_{s}earch$" >< $intent-filter$ >< $actionandroid$: $name$ = "$android.intent.action.SEARCH$"/ >< /$intent - filter$ >

<meta-data android :name=”android.app.searchable” android :resource=”@xml/searchable”/> </activity>

Pour finir, il nous suffit d'écouter toute cette configuration dans notre `Activity`. Dans la méthode `onCreateOptionsMenu(Menu)`, vous allez pouvoir désérialiser le menu que nous avons confectionné plus tôt, récupérer le `MenuItem` de notre recherche et utiliser la classe `MenuItemCompat`, issue de la bibliothèque de compatibilité, pour appeler la méthode `setOnActionExpandListener` et spécifier que nous voulons bien étendre la vue sur toute la largeur de la barre d'action. Le début de la méthode pourrait ressembler à l'exemple suivant :

[Configure le menu de recherche pour l'étendre sur toute la largeur de la Toolbar]java getMenuInflater().inflate(R.menu.menu$_{s}earch, menu$); $MenuItemsearchItem$ = $menu.findItem(R.id.action_{s}earch)$; $MenuItemCompat.setOnActionExpandListener(searchItem$

@Override public boolean onMenuItemActionCollapse(MenuItem item) // Method called when the search view is closed. return true ;) ;

Il est bien inutile de configurer une barre de recherche si nous pouvons pas récupérer le résultat. Pour y parvenir, il suffit de récupérer la `SearchView` à partir du `MenuItem` de la recherche et de lui attacher le *listener* `SearchView.OnQueryTextListener` pour écouter chaque caractère rentrée dans la barre ou quand l'utilisateur soumet la recherche.

[Écoute les caractères rentrées dans la barre de recherche]java final SearchView searchView = (SearchView) searchItem.getActionView() ; searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() @Override public boolean onQueryTextSubmit(String query) // Method called when the search is submitted. querySubmit(query) ; searchView.clearFocus() ; return true ;

@Override public boolean onQueryTextChange(String newText) if (newText == null || newText.length() < 3) return false ; // Method called when the text in the search view changes. return true ;) ;

Dernière chose importante, il faut récupérer la configuration XML de la recherche attachée à notre `Activity` avec notre vue de recherche grâce au `SearchManager`.

[Liaison entre la configuration de la recherche et la vue de la recherche]java final SearchManager searchManager = (SearchManager) getSystemService(Context.SEARCH$_SERVICE$);

Le tout donnera le résultat suivant quand vous cliquerez sur le bouton du menu.

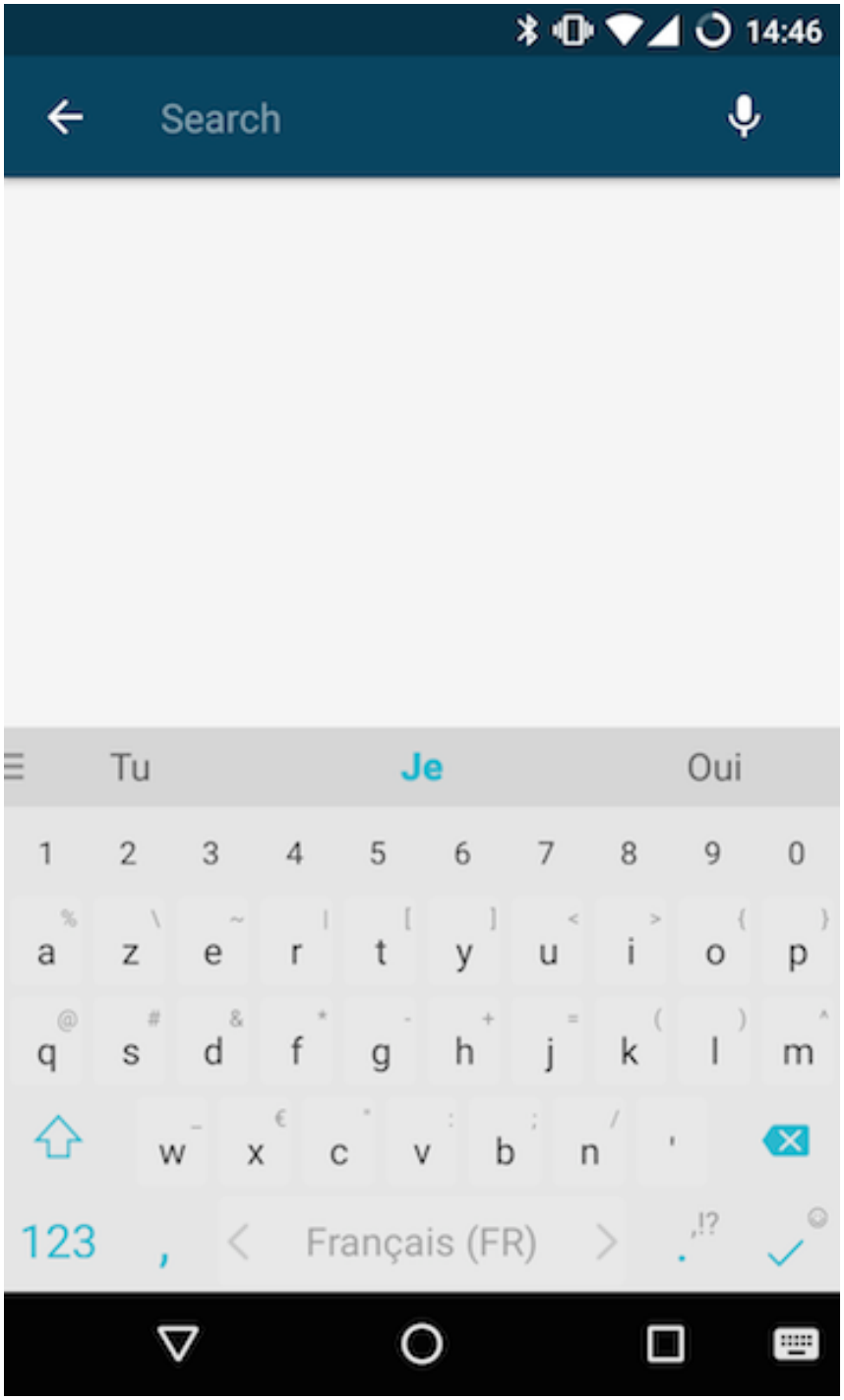


FIGURE 4.5. – Recherche dans la Toolbar

5. Intégrez un menu latéral en Material Design

Ce chapitre aborde plusieurs thématiques divisées en plusieurs sections. La section 1 explique quelle navigation utiliser en fonction de votre situation. Elle ne propose pas une table de correspondance, cela serait impossible d'être exhaustif, mais cela vous fera réfléchir pour pouvoir choisir plus judicieusement la navigation adéquate. La section 2 explique comment l'intégrer dans votre application. La section 3 explique comment placer des *items* statiques dans le menu, qui n'ont pas vocation à changer par la suite. La section 4 explique comment manipuler ces *items* en fonction d'un contexte donné. La section 5 est consacrée à une brève conclusion pour résumer tous les points importants à retenir aussi bien sur la navigation par menu latéral que par le choix de ce type de navigation.

5.1. Intégrer le menu dans son application

Vous êtes peut-être un développeur Android et vous voulez savoir comment passer des anciens menus latéraux aux nouveaux avec le visuel de la version 5 d'Android, le Material Design. Vous ne serez pas déstabilisé puisque le `DrawerLayout` est toujours utilisé. Pour les plus débutants, sachez qu'un `DrawerLayout` est le composant qui vous permet de mettre en place le mécanisme du menu latéral. Il prend deux fils qui correspondent à l'écran principal et au menu latéral.

Avant le Material Design, la mise en oeuvre la plus courante d'un *drawer* était un `FrameLayout` comme écran principal pour y placer des fragments[??] et une `ListView` pour le menu latéral. Avec la venue du Material Design, Google a voulu proposer un nouveau composant pour simplifier la gestion de la liste pour le menu et intégrer automatiquement le rendu Material.

Rappelons que le Material Design est apparu avec la version 5 d'Android. Rendre ce rendu le plus possible compatible avec les vieilles versions permet

une harmonisation des designs à travers les versions des utilisateurs finaux, chose pas forcément possible avant sans un minimum de travail de la part des développeurs.

Commencez par spécifier les dépendances nécessaires dans le fichier build.gradle de votre application ou module :

[Dépendances nécessaires pour le menu latéral et son design]groovy compile 'com.android.support :appcompat-v7 :24.2.0' compile 'com.android.support :support-v4 :24.2.0' compile 'com.android.support :design :24.2.0'

La déclaration d'un écran pour un menu latéral se fait par un *layout* qui contient un `DrawerLayout` comme conteneur racine, un `NavigationView` pour le menu latéral, un `FrameLayout` pour accueillir les fragments du menu latéral (ou depuis d'autres ressources selon les besoins de votre écran) et une `Toolbar` qui va accueillir l'icône dite "hamburger" permettant l'ouverture du menu. Notez quand même qu'il reste possible d'ouvrir le menu d'un simple geste de gauche vers la droite.

[Écran avec un menu latéral]xml <?xml version="1.0" encoding="utf-8" ?> <android.support.v4.widget.DrawerLayout android :id="@+id/drawer$_{l}ayout$"$xmlns$: $android = "http : //schemas.android.com/apk/res/android"xmlns : app = "http : //schemas.android.com/apk/res-auto"xmlns : tools = "http : //schemas.android.com/tools"and$r
$ $layout_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$: $fitsSystemWindows = "true"tools : context = "org.randoomz.demo.design.drawer.DrawerActivi$

<android.support.design.widget.CoordinatorLayout android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight = "match_{p}arent"$ >

<android.support.design.widget.AppBarLayout android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight = "wrap_{c}ontent"android : theme = "@style/ThemeOverlay.AppCompat.Dark.Action$

<android.support.v7.widget.Toolbar android :id="@+id/toolbar" android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight$ = "$wrap_{c}ontent$"$android$: $minHeight$ = "$?attr/actionBarSize"app : layout_{c}ollapseMode = "pin"/ >$

</android.support.design.widget.AppBarLayout>

<FrameLayout android :id="@+id/content" android :layout$_{w}idth$ = "$match_{p}arent$"$android$: $layout_{h}eight = "match_{p}arent"app : layout_{b}ehavior = "@string/appbar_{s}crolling_{v}iew_{b}ehavior"/ >$

</android.support.design.widget.CoordinatorLayout>

<android.support.design.widget.NavigationView android :id="@+id/nav$_{v}iew$"$android$: $layout_{w}idth$ = "$wrap_{c}ontent$"$android$: $layout_{h}eight$ = "$match_{p}arent$"$android$:

$layout_gravity$ = "start"app : $headerLayout$ = "@$layout/view_header$"app : $menu$ = "@$menu/menu_drawer$"/ >< /$android.support.v4.widget.DrawerLayout$ >

Si vous regardez bien le `NavigationView` dans l'exemple précédent, celui-ci renseigne deux attributs intéressants. `app:headerLayout` prend en paramètre un *layout* pour afficher un espace au dessus de la liste du menu déroulant. L'usage usuel consiste à renseigner le profil de l'utilisateur connecté, ou non. `app:menu` renseigne le menu que nous voulons remplir dans le menu. Nous verrons sa déclaration dans les sections suivantes de ce chapitre.

Du côté Java, dans l'activité hôte où vous comptez utiliser le *layout* que vous venez de créer, vous devez initialiser tous ces composants et leurs donner une utilité. C'est ici que les développeurs Android vont avoir de gros changements puisque le menu latéral n'est plus une liste mais un menu, géré comme le menu de la barre d'action sans tous les soucis pour préparer[??] un menu s'il est dynamique. Sans se soucier de savoir comment spécifier les *items* présents dans le menu latéral, il suffit de le récupérer et de lui attacher le *listener* adéquat pour réagir aux *items* voulus.

[Configure la NavigationView dans l'Activity hôte]java public class MainActivity extends AppCompatActivity implements NavigationView.OnNavigationItemSelectedListener private DrawerLayout drawerLayout ; private NavigationView navigationView ;

@Override protected void onCreate(Bundle savedInstanceState) super.onCreate(savedInstanceState) ; setContentView(R.layout.activity$_main$);

final Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar) ; setSupportActionBar(toolbar) ;

drawerLayout = (DrawerLayout) findViewById(R.id.drawer$_layout$);

navigationView = (NavigationView) findViewById(R.id.nav$_view$); $navigationView.setNavigationIte$

@Override public boolean onNavigationItemSelected(@NonNull MenuItem item)
Snackbar.make(drawerLayout, item.getTitle(), Snackbar.LENGTH$_SHORT$).$show$(); $drawerLayout.c$

Cependant, pour permettre de lier l'icône hamburger de la barre d'action, le menu latéral et le `DrawerLayout`, nous devons utiliser une nouvelle classe, `ActionBarDrawerToggle`. Cette classe vient lier ces trois composants pour les faire interagir ensemble et permettre de changer l'état du menu facilement. Pour ce faire, vous devez initialiser cette classe aux côtés du `DrawerLayout`, du `NavigationView` et de la `Toolbar`. Puis, renseigner les méthodes de synchronisation dans trois méthodes de l'`Activity` hôte. Ci-dessous, un exemple concret de sa mise en oeuvre.

[Configuration d'un ActionBarDrawerToggle]java @Override protected void onCreate(Bundle savedInstanceState) super.onCreate(savedInstanceState) ; setContentView(R.layout.ac

// Get and init DrawerLayout and Toolbar. toggle = new ActionBarDrawerToggle(this, drawerLayout, toolbar, R.string.drawer$_o$*pen*, *R.string.drawer*$_c$*lose*); *drawerLayout.addD*

@Override public boolean onOptionsItemSelected(MenuItem item) if (toggle.onOptionsItemSelected(item)) return true ; return super.onOptionsItemSelected(item) ;

@Override public void onConfigurationChanged(Configuration newConfig) super.onConfigurationChanged(newConfig) ; toggle.onConfigurationChanged(newConfig) ;

@Override protected void onPostCreate(Bundle savedInstanceState) super.onPostCreate(savedInstanceState) ; toggle.syncState() ;

Vous remarquerez que l'activité hôte n'étend pas `FragmentActivity` ou `ActionBarActivity` mais `AppCompatActivity`. Cette nouvelle classe est issue de la bibliothèque de compatibilité appcompat-v7 et demande beaucoup de changement dans votre projet notamment dans vos styles et dans l'utilisation de la barre d'action. Toute son utilisation est hors scope de ce tutoriel et en attendant un tutoriel sur Zeste de Savoir, consultez sa documentation pour voir comment l'utiliser. Souvenez vous aussi que tous les exemples donnés dans ce tutoriel sont issus d'une application open source développée spécialement pour ce tutoriel et disponible sur ce projet GitHub . Cela pourrait vous aider à savoir comment gérer vos styles et comment manipuler basiquement la barre d'action avec cette bibliothèque.

5.2. Menu statique

A partir de cette étape, si vous testez votre application, vous devriez avoir un écran (vide si n'avez attaché aucun fragment) et un menu latéral vide qui s'affiche

5. Si vous ne savez pas ce que c'est un fragment, je vous redirige vers un autre tutoriel Android qui vous explique tout ce qu'il faut savoir à ce sujet : https ://zestedesavoir.com/tutoriels/278/aller-plus-loin-dans-le-developpement-android/

6. Retrouvez l'annonce officiel sur le blog des développeurs Android : http ://android-developers.blogspot.fr/2014/12/android-studio-10.html

7. Préparer un menu donne la possibilité aux développeurs d'activer/désactiver ou modifier dynamiquement un menu présent dans la barre d'action.

lorsque vous cliquez sur l'icône en haut à gauche de l'écran ou lorsque vous venez le chercher avec votre doigt sur le bord gauche de l'écran en le glissant vers la droite.

Vous pouvez faire ce que vous voulez de l'écran principal, il ne nous intéresse pas pour ce tutoriel. Par contre, le menu latéral ne va pas rester vide bien longtemps. Nous allons voir la façon "simple" pour ajouter des *items* dans ce menu. Par simple, j'entends un menu statique pas spécialement voué à changer pendant l'exécution de l'application.

Si vous êtes un développeur Android et que vous avez bien lu toutes mes explications, vous devriez savoir quoi faire. Nous allons simplement créer un fichier XML avec tous les *items* de notre menu et nous allons l'attacher à notre menu latéral dans son fichier XML. D'ailleurs, pour ce dernier point, c'est déjà fait. Rappelez-vous, nous avons rajouté un schéma par lequel nous pouvons y accéder via `app` . Puis, nous l'avons utilisé dans `NavigationView` avec son attribut `menu`, ce qui donnait la déclaration suivante : `app:menu="@menu/drawer_view"`. Il nous reste donc à créer ce fichier `drawer_view` dans le dossier `menu` de votre projet.

Inutile de vous expliquer comment créer des menus, ils ne diffèrent en rien des autres menus d'Android depuis sa version 1. Si vraiment vous ne savez pas comment les créer, allez jeter un oeil à d'autres tutoriels comme ce chapitre du tutoriel Android pour les débutants sur Zeste de Savoir.

[Déclaration d'un menu statique]xml < ?xml version="1.0" encoding="utf-8" ?> <menu xmlns :android="http ://schemas.android.com/apk/res/android"> <group android :checkableBehavior="none"> <item android :id="@+id/menu$_a$dd"$android$: $icon$ = "@$android$: $drawable/ic_menu_add$"$android$: $title$ = "@$string/menu_add$"/ >< $itemandroid$: id = "@+$id/menu_delete$"$android$: $icon$ = "@$android$: $drawable/ic_menu_delete$"$android$: $title$ = "@$string/menu_delete$"/ >< /$group$ >< /$menu$ >

Ce qui est intéressant de constater, c'est le visuel ! Automatiquement, le menu latéral va rajouter nos 2 *items* dans le menu latéral. Tout se fait en interne de la bibliothèque pour vous simplifiez la vie. N'hésitez pas à jouer avec ses menus pour obtenir le rendu qui vous convient le mieux !

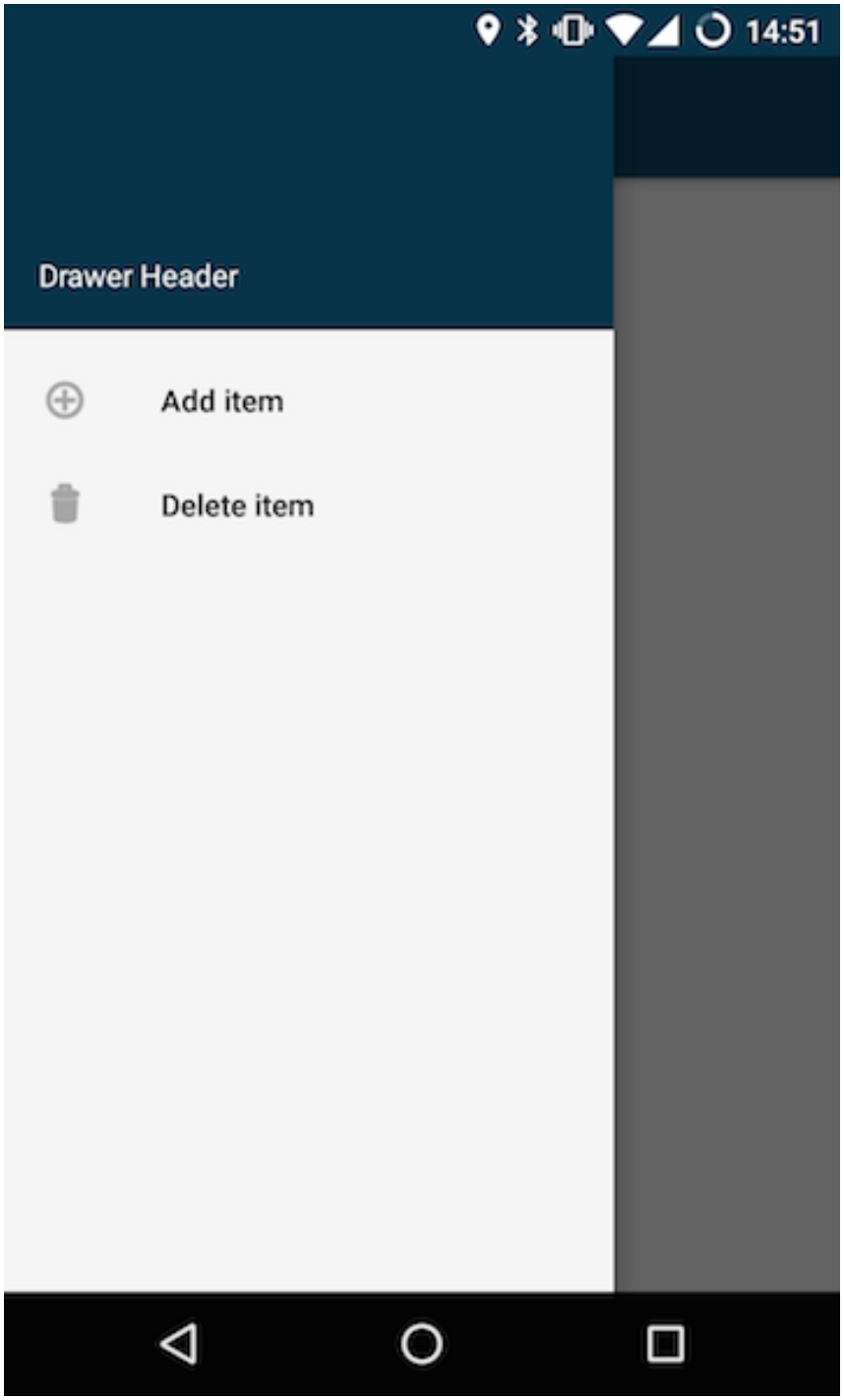


FIGURE 5.1. – Exemple d'un menu statique

5.3. Menu dynamique

Dans 90% des cas, vos besoins se limiteront à des menus statiques mais il peut arriver que vous ayez besoin de modifier dynamiquement les *items* du menu latéral. Dans ce cas, il faudra utiliser l'API du composant `NavigationView` consacrée au menu.

Cela serait trop ambitieux de vous expliquer tout ce qu'il est possible de faire puisque la bibliothèque pourrait changer quelques contrats dans son utilisation par rapport au moment où j'ai écrit ces lignes et les possibilités sont bien trop grandes et trop peu intéressantes pour toutes les énumérer. Nous allons donc procéder par l'exemple afin que vous compreniez les mécanismes de cette API et que vous soyez alors capable de les adapter à votre cas d'usage.

Qu'est ce que nous voulons faire ? Nous allons permettre à l'utilisateur de rajouter ou supprimer des *items* et de les ranger dans une section dédiée du menu latéral. Visuellement, nous voulons une nouvelle section avec un titre et dedans toutes les *items* de l'utilisateur.

Nous devons faire évoluer le fichier XML du menu pour rajouter une section et un menu :

[Déclaration du menu dynamique]xml <?xml version=”1.0” encoding=”utf-8” ?> <menu xmlns :android=”http ://schemas.android.com/apk/res/android”> <!– Previous menu. –>

<item android :id=”@+id/menu$_{d}ynamic$"$android : title$ = "@$string/menu_{d}ynamic$" >< $menu$ >< $/menu$ >< $/item$ >< $/menu$ >

Notez bien l'*item* avec un identifiant. Cet identifiant est nécessaire pour le récupérer dans le code Java et pour le manipuler. Du côté Java, nous allons réagir aux boutons du menu que nous avons rajouté dans la section précédente. Nous devons rajouter du comportement dans la méthode `onNavigationItemSelected(MenuItem)` pour créer à la volée des *items* supplémentaires.

[Rajoute ou supprime dynamiquement des items du menu latéral]java @Override public boolean onNavigationItemSelected(@NonNull MenuItem item) final MenuItem dynamicItem = navigationView.getMenu().findItem(R.id.menu$_{d}ynamic$); $finalSubMenusubMenu =$ $dynamicItem.getSubMenu()$; $switch(item.getItemId())caseR.id.menu_{a}dd : subMenu.add(Menu.$

Ce que nous avons fait est très simple. Avec l'instance du menu latéral, nous avons été cherché la nouvelle section par l'identifiant de l'*item* que nous avons spécifié dans le fichier XML. Puis, nous avons récupéré le menu que contenait la section.

Puis, nous ajouter ou supprimons un menu en fonction de ce que l'utilisateur désire faire.

Finalement, nous obtenons un menu latéral avec un menu dynamique. C'est aussi simple que cela et pour les développeurs Android expérimentés, vous remarquez que l'API est semblable à celle de toutes les autres API sur les menus.

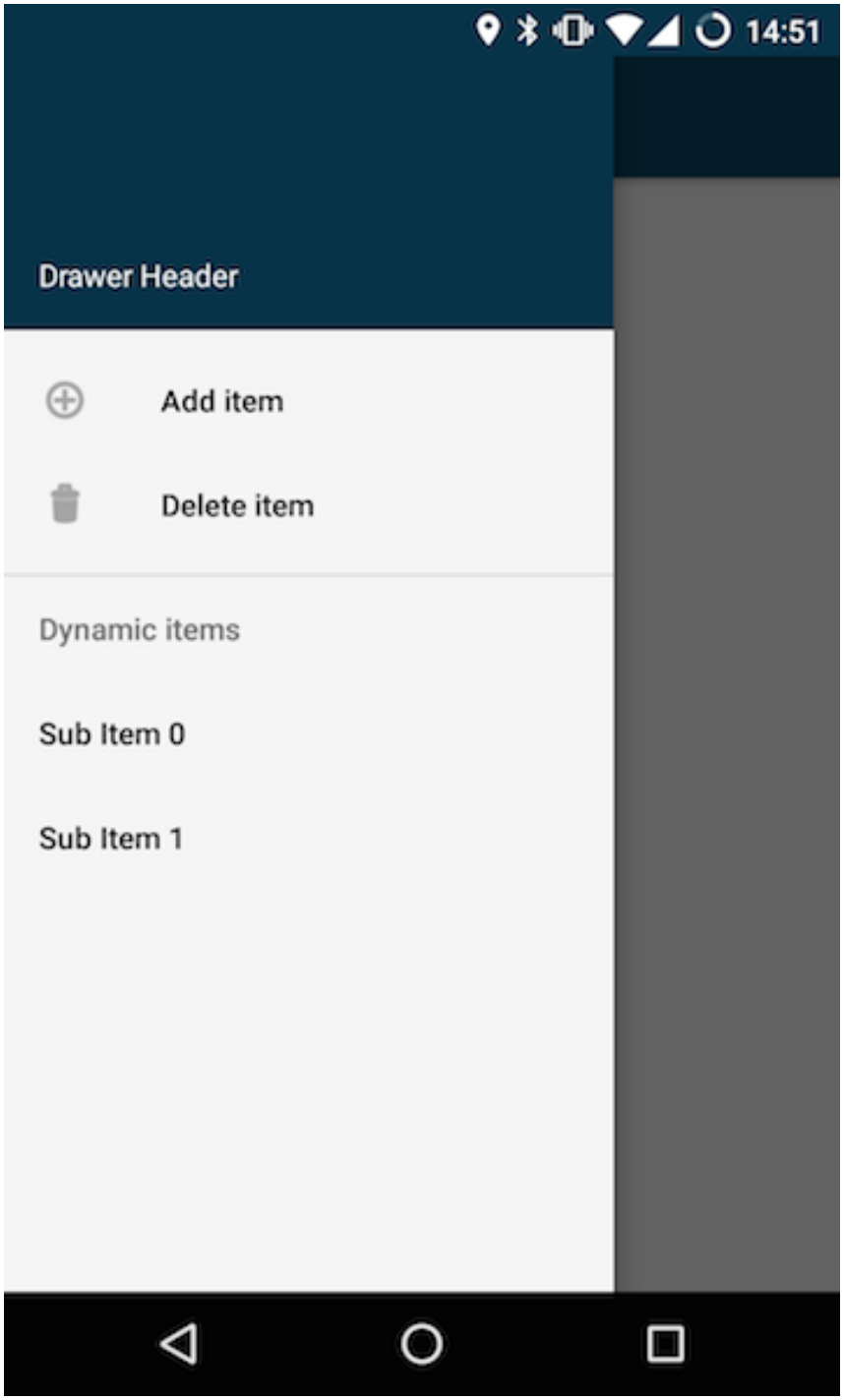


FIGURE 5.2. – Exemple d'un menu dynamique

Nous voici à la fin de ce chapitre. Alors, qu'est-ce que vous avez appris ? Malgré le fait que cela soit un modeste mini tutoriel, pas mal de choses. Déjà, vous êtes capable d'intégrer un menu latéral dans votre application ou de mettre à jour votre menu latéral de la liste vers le `NavigationView`. Vous êtes aussi capable de créer des menus statiques et dynamiques avec ce nouveau composant et donc de vous adapter à toutes les situations. Rien ne vous empêche de concevoir un menu hybride avec du statique et du dynamique, la seule barrière est votre imagination.

Mais plus important, vous avez compris pourquoi et dans quelle situation vous devez utiliser un menu latéral. Pour rappel, un menu latéral est utilisé pour des applications avec des parties distinctes qui n'ont pas forcément de liens entre elles. Vous avez pu le constater avec les exemples mentionnés dans ce tutoriel (repris du projet open source AdvancedAndroidDevelopment sur GitHub) qui possédaient un menu latéral avec différentes sections sans aucun lien entre elles, tout en restant pertinent de les rassembler dans un seul et même composant de navigation.