1. Découverte des ressources

On va partir d'un projet minimal, type « Empty Views Activity », nommez-le `TP3`. Aujourd'hui, nous allons étudier les layouts et les vues.

1.1. Ressources

Ce sont les fichiers XML qui sont dans *projet*`/src/main/res`. Il y en a de plusieurs sortes, dont :

- `res/values` : le fichier `strings.xml` contient les textes de votre application : labels, messages, etc. Ces textes sont identifiés par l'attribut `name` et référencés dans les layouts par `@string/nom`, et par `R.string.nom` dans les sources java. Voir le TP2 pour un exemple de traduction des textes.
- `res/drawable` et `res/mipmap`, ce sont des images, icônes et autres dans plusieurs résolutions. On ne s'y intéressera pas cette semaine.
- `res/layout` : ce sont les définitions d'écrans des activités, par exemple `activity_main.xml`. Les balises XML et leurs attributs définissent les éléments affichés : boutons, zones de saisie, etc. Dans le TP3, on va se concentrer sur ces fichiers.

Remarque : les fichiers XML peuvent être affichés soit sous forme d'un formulaire ou de manière graphique, soit sous forme XML brute. Voyez l'onglet en haut à droite de l'éditeur, ex: `Code`, `Split` ou `Design`. Le mode Code offre un contrôle total. Il est préférable au mode graphique quand on sait ce qu'on fait.

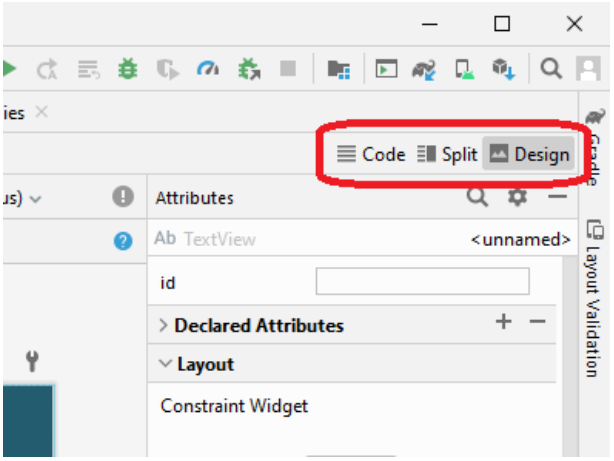

Figure 1: Mode Code ou Design

1.2. Ressources, identifiants et références

Avant de commencer, quelques notions très importantes. La plupart des éléments des ressources sont identifiés. Par exemple, tous les textes présents dans `res/values/strings.xml` :

```
<string name="identifiant">texte</string>
...
```

L'attribut `name` définit **l'identifiant de la ressource chaîne**.

Dans le TP2, vous aviez brièvement vu que, dans le fichier `res/layout/activity_main.xml`, le texte avait été remplacé par ceci :

```
<TextView android:text="@string/identifiant"
    ... />
```

La balise `TextView` représente un simple texte affiché sur l'écran. Le texte affiché est défini par l'attribut `android:text`. La syntaxe `@string/identifiant` est une **référence de ressource chaîne**. Il y a aussi des références de couleurs, par exemple `@color/teal_700`, de style, etc.

Ne confondez pas *identifiant* (= définition) et *référence d'identifiant* (= utilisation).

Dans certains layouts, il faut identifier les vues. Ça consiste à rajouter un attribut spécial `android:id="@+id/nom"` :

```
<TextView
    android:id="@+id/titre"
    ... />
```

La valeur d'attribut `@+id/titre` définit l'identifiant « titre » attribué au `TextView`.

Ensuite, dans une autre vue, on peut référencer cet identifiant avec une syntaxe subtilement différente, sans le `+` :

```
<Button
    android:layout_below="@id/titre"
    ... />
```

Se souvenir que :

- `@+id/nom` définit un identifiant unique,
- `@id/nom` est une référence à cet identifiant.

1.3. Ressources et classe R

Une interface utilisateur Android est définie par un fichier XML placé dans `res/layout`. Le nom de ce fichier est une sorte d'identifiant pour l'interface qu'il définit. Regardez dans `MainActivity.java`, il y a une ligne `setContentView(R.layout.activity_main);`. C'est cette instruction qui indique à l'activité quelle est l'interface qu'elle doit afficher, celle de `activity_main.xml`.

La syntaxe `R.layout.nomdulayout` est très particulière. Il faut comprendre que `R` est une classe générée automatiquement à partir de tout ce qui est trouvé dans les ressources. Voici comment elle serait programmée en Java :

```
public final class R {
    public static final class string {
        public static final int app_name=0x7f080000;
        public static final int titre=0x7f080001;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
```

```
    public static final class menu {
        public static final int main_menu=0x7f050000;
        public static final int context_menu=0x7f050001;
    }
    ...
```

Cette classe `R` associe des entiers à chaque ressource identifiée : chaînes, interfaces, éléments d'interface, icônes, etc, chacun dans une catégorie. Ainsi tous les layouts de `res/layout/*` sont dans `R.layout.*`, tous les identifiants de chaînes de `res/values/strings.xml` sont dans `R.string.*`, etc.

Avant, il était possible de visualiser cette classe, mais ce n'est plus possible depuis peu. La classe `R` est générée de manière invisible par AndroidStudio.

2. Layout et vues

On va maintenant approfondir l'étude des layouts.

Un layout contient, sous forme de balises XML la description de l'interface, c'est à dire les « vues » ou composants d'interface à mettre dedans et comment les placer les unes par rapport aux autres.

2.1. Arbre des vues

Il y a deux types de vues.

- Les « vues simples », comme `TextView`, `Button`, `CheckBox`, etc. Il y en a un grand nombre. Ces vues sont simples parce qu'elles ne contiennent aucune autre vue à l'intérieur.
- Les « vues de rangement », comme `LinearLayout`, `ConstraintLayout`, etc. Au contraire des vues simples, les vues de rangement contiennent d'autres vues à l'intérieur, et leur but est par exemple de les aligner automatiquement, ou selon des critères dynamiques.

Toutes ces vues d'une interface forment un arbre (*Component Tree*) dont les branches sont les vues de rangement, et les feuilles sont les vues simples.

Voici l'arbre de l'interface du TP2 :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    ----
    android:orientation="vertical">

    <TextView
        android:id="@+id/textview"
        ---/>

    <Button
        ---
        android:text="@string/do_round_robin"
        android:onClick="onPermuter"/>
</LinearLayout>
```

La racine de l'arbre est un `LinearLayout` vertical, il contient deux feuilles, un `TextView` et un `Button`.

La figure 2 montre les trois vues, un `LinearLayout` qui contient un `TextView` et un `Button`.

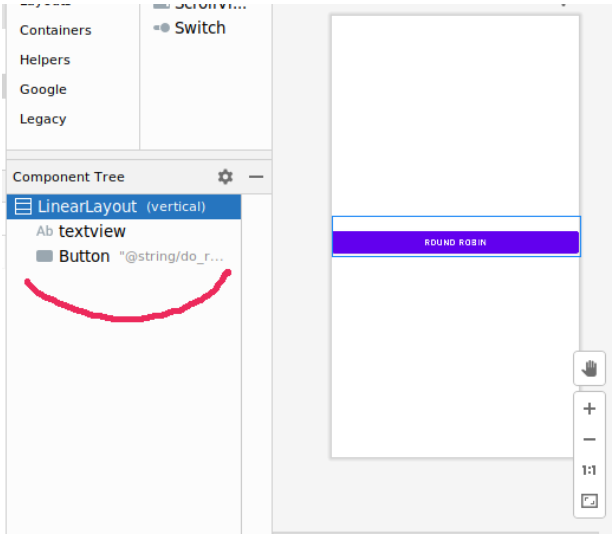

Figure 2: Arbre de vues

2.2. Paramétrage des vues

Les vues d'une interface ont de très nombreuses propriétés : titre, couleur, taille, etc. Ces propriétés peuvent être spécifiées par programmation, à l'aide de *setters*, mais elles sont aussi définies dans le fichier XML sous forme d'attributs.

```
<TextView
    android:id="@+id/textview"
    ---
    android:textSize="28dp"
    android:textColor="@color/teal_700"
    android:text="Bonjour"
    android:padding="10dp"
    android:gravity="center"/>
```

Il faut du temps pour apprendre toutes les propriétés utiles pour construire des interfaces agréables.

2.3. Attributs obligatoires

Il y a seulement deux attributs absolument obligatoires pour chacune des vues de l'interface, `android:layout_width` et `android:layout_height`. Ces deux propriétés définissent la taille souhaitée à l'écran. Le layout est invalide s'il manque ces informations sur l'une des vues, y compris la vue racine.

Ces deux informations, largeur et hauteur, peuvent avoir ces valeurs :

- `match_parent` : signifie que la vue souhaite adopter la taille (largeur ou hauteur) de l'élément de rangement parent,

- `wrap_content` : signifie que la vue veut être la plus ajustée possible,
- `NNNdp` : la vue veut une taille de *NNN* pixels. Les `dp` sont des pixels dont la taille est indépendante de l'écran. 1 dp = 1 pixel sur un écran 160 dpi. Si l'écran est en 320 dpi, alors 1 dp = 2 pixels. C'est pour éviter d'avoir des interfaces qui changent de taille d'un smartphone à l'autre, selon la finesse des pixels de l'écran.
- `0dp` : signifie que la place attribuée à la vue par son parent dépendra des autres vues et d'un autre attribut appelé `android:layout_weight`. Voir les exercices plus loin.

2.4. Composants à connaître

Android contient plusieurs dizaines de composants d'interface :

- des vues : `TextView`, `Button` et `EditText` ...
- des groupes : `LinearLayout`, `TableLayout`... lire declaring-layout.html.

Il faut savoir que les composants évoluent et peuvent être déclarés obsolètes, par exemple `DigitalClock`, `Gallery`...

On vous invite à explorer ces documentations, avec curiosité.

3. Tutoriel sur les groupements de vues

On va s'intéresser aux vues qui permettent d'arranger d'autres vues. Elles dérivent de la classe `ViewGroup`. Pour simplifier, elles ont des vues enfants et décident de leur placement, taille et position, en fonction de propriétés présentes sur les enfants. Voici les quatre à connaître :

- `LinearLayout` pour former des lignes ou des colonnes, voir doc
- `TableLayout` pour disposer en tableau, voir doc
- `ConstraintLayout` pour des dispositions dynamiques complexes.

3.1. Mise en page avec des LinearLayout

C'est le composant de rangement le plus simple et le plus facile à employer. Il dispose ses vues enfants soit en ligne, soit en colonne. Il n'est conseillé que pour des interfaces simples.

Téléchargez ce document XML et appelez-le `res/layout/linear.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:text="OK"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:text="Annuler"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"/>
    <Button
        android:text="Annuler tout"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Examinez la mise en page en comparant avec les propriétés des vues, puis faites les changements suivants :

- Mettez `match_parent` pour `android:layout_width` de l'un des boutons, puis annulez le changement.
- Mettez `match_parent` pour `android:layout_height` du deuxième bouton, regardez l'effet, puis annulez. Vous avez vu que le 3e bouton n'est plus visible.

Retenir : `match_parent` signifie « prendre toute la place restante » et `wrap_content` signifie « prendre la place minimale ».

- Mettez `200dp` puis `300dp` pour `android:layout_height` du deuxième bouton, regardez son effet, puis annulez.
- Changez `android:orientation` en `horizontal` pour le `LinearLayout`, laissez ce changement. Passez en mode paysage si les vues sont trop encombrantes.

On voit que les vues ont une largeur dépendant de leur contenu. On voudrait qu'elles aient la même largeur.

- Ajoutez `android:layout_weight="1"` pour chacune. C'est pas très visible, mais la largeur n'est pas exactement la même à cause des titres.
- Changez `android:layout_width="0dp"` pour les trois vues. C'est ce qui donne le meilleur résultat : largeur nulle combinée avec un poids non nul.
- Changez `android:layout_width="wrap_content"` pour le `LinearLayout` puis annulez ce changement.

Maintenant, à vous. Vous devez obtenir ceci :

Figure 3: Saisie d'un mot de passe

1. Copiez `linear.xml` et appelez-le `password.xml`
2. Supprimez les 2e et 3e boutons, ne laissez que le `OK`.
3. Ajoutez un `EditText` devant le bouton `OK`.
4. Remplacez la propriété `android:text` par `android:hint` et mettez la chaîne `"mot de passe"`.

5. Puis modifiez les largeurs pour obtenir le résultat demandé. C'est à dire que l'`EditText` prend presque toute la place en largeur, tandis que le bouton `OK` est le plus petit possible. Pour cela, il faut mettre un poids nul au bouton `OK` mais aussi lui donner une taille minimale.

Dans un cas général, on est amené à combiner des `LinearLayout` horizontaux et verticaux. Voici un exemple.

1. Copiez `password.xml` et appelez-le `calendar.xml`
2. Faites en sorte d'ajouter un `CalendarView` au dessus des deux vues `EditText` et `Button` pour obtenir l'image ci-dessous. Pour y arriver, vous devrez définir deux `LinearLayout`, un horizontal pour les boutons, contenu dans un vertical. Vous devrez aussi jouer sur les poids et les hauteurs pour garantir une visibilité et une taille à tout le monde.

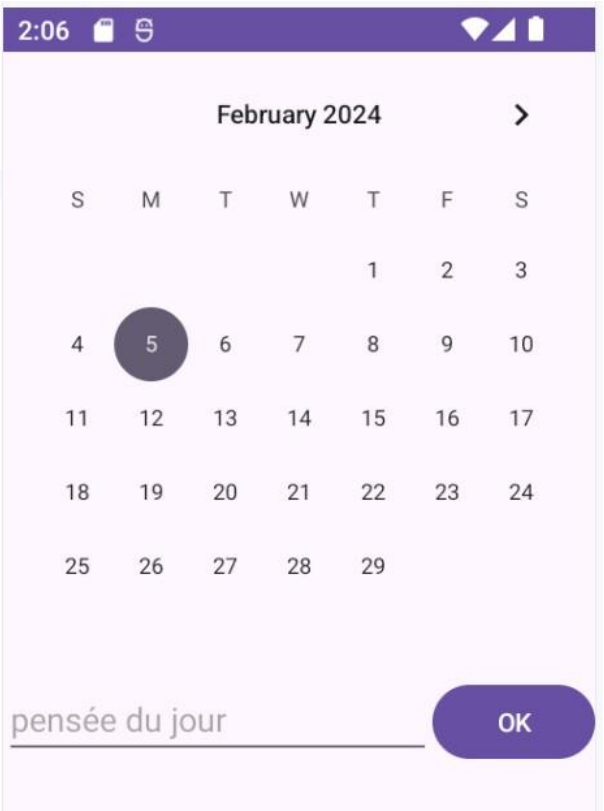

Figure 4: Calendrier et mot du jour

Vous allez recevoir un avertissement sur le coût des calculs en cas d'imbrication des layouts portant des poids. *Nested weights are bad for performance*. La présence de poids oblige l'algorithme du layout à faire plusieurs parcours de l'arbre des vues pour définir toutes les tailles correctement. Mais ici, il y a très peu de vues, ce n'est pas gênant.

3.2. Mise en page avec un TableLayout

Ce type de vue sert à organiser les éléments comme dans un tableau HTML. Ce sont les mêmes notions codées en XML. Voici un exemple à télécharger sous le nom `res/layout/tableau.xml`

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

    <TableRow>
        <Button android:text="1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <Button android:text="col2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
    </TableRow>

    <TableRow>
        <Button android:text="2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <Button android:text="col2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
    </TableRow>

</TableLayout>
```

Par manque de place, je n'ai pas pu mettre trop de lignes et de colonnes, mais rajoutez par copie-collage une colonne de plus, puis deux ou trois lignes. Regardez ce que ça fait dans l'aperçu.

Figure 5: Table

Par défaut, dans une colonne toutes les cases font la même largeur, et même si une colonne est plus étroite, elle reste quand même assez large. Il n'y a pas grand chose à configurer pour changer l'apparence à part deux propriétés :

- Ajoutez `android:shrinkColumns="0"` dans la balise `<TableLayout>` en haut. La première colonne 0 est autorisée à devenir plus étroite si la place manque, ou carrément disparaître. Pour voir cela, allongez le texte de l'un des boutons 2e colonne.
- Ajoutez `android:stretchColumns="1"` dans la balise `<TableLayout>`. La 2e colonne va s'élargir autant que possible – allongez le texte d'un bouton. Plusieurs colonnes peuvent être étirées : `android:stretchColumns="1,2"`.

3.3. Mise en page avec un ConstraintLayout

Ce type de `ViewGroup` permet de mettre à jour la disposition de manière dynamique et polyvalente. Le principe est de dire pour chaque vue, où on doit la mettre sur l'écran : en dessous de telle autre, à droite de telle autre. . . On définit donc un ensemble de contraintes de placement qu'Android essaye de satisfaire. Paradoxalement, un `ConstraintLayout` est conseillé pour des interfaces complexes, parce que les autres layouts sont beaucoup plus lents pour faire la mise en page.

Plus précisément, dans un `ConstraintLayout`, on doit spécifier chaque bord d'une vue. Les bords sont le haut, le bas, le côté gauche et le côté droit, `Top`, `Bottom`, `Left`, `Right`. Si on spécifie, par exemple, le côté gauche et la largeur, alors le côté droit s'en déduit. Donc par exemple, on peut indiquer la hauteur et la largeur avec `wrap_content` et seulement indiquer sur quoi sont alignés les bords haut et gauche d'une vue.

Les alignements des côtés sont spécifiés avec les attributs des vues. Ce qui fait peur au début, c'est qu'il y a 4 à 6 attributs pour chaque vue. Au minimum, il y a les alignements gauche et haut avec la largeur et hauteur. Au maximum, il y a les alignements de tous les côtés et en plus les largeurs et hauteurs valant `0dp`.

Heureusement, ça devient plus simple une fois qu'on a un peu joué avec.

Pour commencer, la taille d'une vue peut être minimale : `wrap_content`, maximale : `match_parent` ou donnée par les contraintes de placement : `0dp`.

Ensuite, les attributs pour spécifier les alignements ont des noms compliqués. La forme générale des contraintes est `app:layout_constraintXXX_toYYYOf` avec *XXX* et *YYY* valant `Top`, `Bottom`, `Left`, `Right`. Cela donne des attributs comme `app:layout_constraintTop_toTopOf`, `app:layout_constraintRight_toLeftOf`, etc.

Par exemple, `app:layout_constraintLeft_toRightOf` veut dire d'aligner le côté gauche de cette vue sur le côté droit d'une autre vue. L'autre vue est indiquée par son identifiant. Par exemple, on peut écrire `app:layout_constraintTop_toBottomOf="@id/calendrier"` pour placer cette vue en dessous du calendrier. Quand l'autre vue est l'écran lui-même, alors on met une valeur spéciale : `"parent"`.

Voici maintenant un exemple à télécharger sous le nom `res/layout/constraint.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
```

```
        android:id="@+id/button1"
        android:text="BTN1"
        android:backgroundTint="#ff683e"
        app:cornerRadius="8dp"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_width="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>
    <Button
        android:id="@+id/button2"
        android:text="BTN2"
        android:backgroundTint="#19C7C1"
        app:cornerRadius="8dp"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/button1"
        android:layout_width="0dp"
        app:layout_constraintLeft_toLeftOf="@id/button1"
        app:layout_constraintRight_toRightOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Le premier bouton est positionné en haut. Le second bouton est mis sous le premier, et comme sa largeur est nulle, elle est forcée à ce que donnent ses contraintes : du bord gauche du bouton 1 au bord droit du parent.

À vous maintenant. Essayez de reproduire cette disposition avec un `ConstraintLayout` :

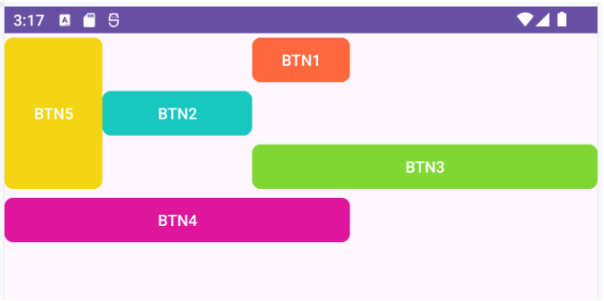

Figure 6: Résultat à obtenir avec `constraint.xml`

Le bouton 1 ne change pas. Le bouton 2 est placé en bas et à gauche du n°1 et son côté gauche est aligné avec le côté droit du bouton 5 (le bouton 2 s'étire horizontalement). Le bouton 3 est sous le n°2 et va horizontalement du bord droit du bouton 2 au bord droit de la fenêtre. Et ainsi de suite. Vous constatez que le bouton 5 est référencé par le bouton 2 mais il est créé après. Et le bouton 5 a besoin du bouton 3 qui dépend du 2. C'est le mécanisme de satisfaction des contraintes qui résoud ce genre de boucle.

Pour positionner une vue à une certaine distance d'une autre, on utilise les attributs `layout_marginXXX`. Deux autres contraintes existent : `layout_constraintZZZ_bias` avec

ZZZ étant `Vertical` ou `Horizontal`. Elles permettent de positionner une vue de manière proportionnellement à l'espace disponible – elles déséquilibrent une contrainte en faveur de l'un ou l'autre côté. La valeur à mettre est un réel entre 0 et 1, par défaut, c'est *0.5*.

4. Exercices

Chacun de ces exercices consiste à créer un fichier `res/layout/`*exercice*`.xml` en mettant le titre de l'exercice à la place. Il vous suffit de changer l'appel à `setContentView(R.layout.exercice)` dans `MainActivity.java` pour passer d'un exercice à l'autre.

Par exemple, vous créez un nouveau fichier `res/layout/sport.xml` et vous mettez `setContentView(R.layout.sport)` dans la méthode `onCreate` de `MainActivity`. C'est ce layout qui sera affiché.

Utilisez le dessinateur (onglet *Design*) pour ajouter les vues, mais surveillez de temps en temps ce qui est généré en XML. Vous devrez apprendre à vous servir du dessinateur : gérer l'imbrication des vues et des containers, modifier les propriétés des vues, etc.

Dans les exercices, on demande de créer au minimum un écran qui permette de faire l'opération indiquée et au mieux, si vous avez le temps, cet écran sera ergonomique et élégant quelle que soit la taille et l'orientation de la tablette : les vues bien alignées, avec un libellé en face, etc.

Également : au minimum, les textes seront en dur dans le layout, et au mieux ils seront sous forme de ressources dans `strings.xml`.

4.1. Trombinoscope : trombi.xml

Concevoir un écran permettant d'afficher l'un(e) des étudiant(e)s d'une promotion. Par exemple :

- Le groupe (centré horizontalement),
- le nom, le prénom sous forme de deux zones de texte côte à côte,
- la photo (la plus grande possible à l'écran).
- un bouton pour envoyer un message.

Concernant l'image, vous pouvez télécharger une image png ou webp libre de droits et acceptable, et la placer dans les ressources, voici la démarche :

1. Téléchargez votre image.
2. Ouvrez le menu `View`, puis `Tools Window`, puis `Resource Manager`. Il apparaît dans le panneau de gauche.
3. Cliquez sur le bouton + qui est tout en haut à gauche du panneau, puis l'item du bas `Import Drawables`.
4. Naviguez et choisissez le fichier image voulu.
5. Cliquez sur `Next` puis `Import`.
6. Il reste à l'afficher dans l'`ImageView` : `app:srcCompat="@drawable/nom_image"`.

4.2. Saisie d'un voyage en avion : avion.xml

Concevoir un écran permettant de saisir quelques informations sur un vol d'avion. Par exemple :

- La compagnie aérienne (une ligne de texte),

- l'aéroport de départ, l'aéroport d'arrivée (deux zones de texte côte à côte),
- la date JJ/MM/AAAA (ne mettez pas un `CalendarView`, il est trop grand, mais un texte),
- l'heure HH:MM (texte),
- une case à cocher Aller ou Retour (c'est le vol retour si elle est cochée),
- trois boutons exclusifs pour la classe : économique, affaires et première. Ce sont des `RadioButton` qu'il faut placer dans un même `RadioGroup`.
- une barre d'appréciation (`RatingBar`) avec 5 étoiles pour noter la qualité du vol.

Le `RatingBar` est fait pour attribuer une note sous forme d'étoiles, mais il est un peu déroutant à utiliser :

- Ne lui imposez pas une largeur autre que `"wrap_content"` sinon le nombre d'étoiles dépendra de la largeur effective,
- Pour fixer le nombre d'étoiles, affectez sa propriété `android:numStars`, mettez-en 5,
- Vous pouvez aussi fixer le pas, c'est à dire la granularité des notes, affectez par exemple `android:stepSize="0.5"` qui signifie qu'on peut mettre des notes de 0.5 en 0.5.

Pour les textes à saisir, vous veillerez à définir des `inputType` appropriés.