Dev mobile 101 : Les bases du développement mobile

Cette première formation a pour but de donner les bases de développement mobile.

Compétences abordées : JS, React Native, Expo

Prérequis

Pour suivre cette formation, il faut déjà avoir des bases en DevWeb (HTML, CSS, JS). Nous vous conseillons de suivre les formations Devweb 101 : Les bases du développement web et Devweb 104 : Initiation au Javascript & à JQuery.

Installation

Installation des logiciels

Pour cette formation, il faut installer la version LTS de Node.js ainsi qu'un IDE. Je vous conseille WebStorm, gratuit avec votre mail Centrale, mais vous pouvez utiliser celui que vous voulez.

Nous allons utiliser Expo pour compiler l'application et il faudra donc installer l'application sur votre téléphone pour voir les changements en temps réel :

- Expo pour Android
- Expo pour iOS

Initialisation de l'application

Afin de créer l'application, vous allez taper la commande suivante dans votre cmd ou terminal en remplaçant NomDuProjet par ce que vous voulez.

```
npx create-expo-app@latest
```

En cas d'erreur :

```
File C:\Users\mimmo\AppData\Roaming\npm\expo.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see about_Execution_Policies at https:/go.microsoft.com/fwlink/?LinkID=135170.
```

Il faut lancer la commande

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

```
Set-ExecutionPolicy Unrestricted -Scope CurrentUser
```

Ouvrez votre projet dans votre IDE et dans le terminal, tapez :

```
npm start
```

Une page devrait s'ouvrir avec un QR code.

- Pour iOS : scannez le QR code avec l'appareil photo
- Pour Android : scannez le QR code depuis l'application Expo

Il faut que votre ordinateur et votre téléphone soient connectés au même réseau Wifi.

Félicitations, vous venez de créer votre première application mobile !

Modifier l'application

Le fichier App.js est l'équivalent du index.html en développement web. C'est ce fichier qui sera exécuté au lancement de l'application.

Structure du code

Actuellement dans le App.js, nous pouvons trouver ce code :

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
```

```
});
```

Et vous devriez voir ceci sur votre téléphone :

Déjà, nous remarquons que le code est divisé en trois parties :

- Les import : on importe les librairies et éléments dont on a besoin
- Le contenu de la page
- La StyleSheet (qui correspond au CSS en développement web)

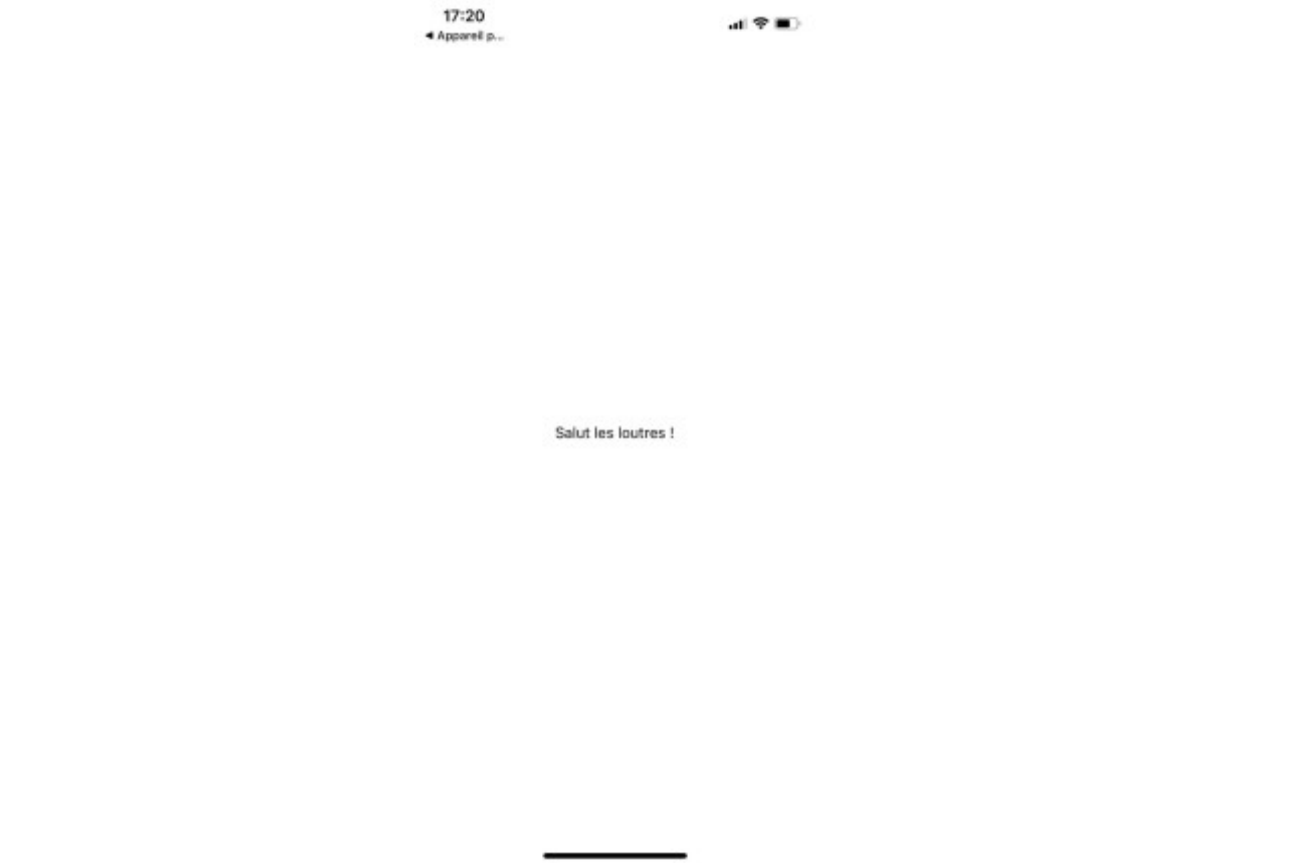
Contenu de la page

Comme vous pouvez le voir, le contenu de la page marche, comme en HTML, avec des balises. C'est du JSX.

Tout d'abord, il y a la balise <View> qui contient tout le contenu. Cette balise correspond à la en HTML. Dans la balise View, nous trouvons la balise Text (je ne pense pas avoir besoin d'expliquer ce qu'elle fait, mais c'est la balise pour mettre du texte) et la balise StatusBar. Cette dernière met en forme la barre de statut de votre téléphone, elle n'est pas très importante.

Vous pouvez changer le contenu de la balise Text pour commencer.

Il suffit d'enregistrer pour que l'application se mette à jour sur votre téléphone.



StyleSheet

Comment en CSS, il y a plusieurs façons pour changer le style d'un élément.

La première façon (déconseillé) est de le mettre dans l'attribut style de cette façon :

```
<Text style={{color: 'red'}}>Salut les loutres !</Text>
```

Comme vous pouvez le voir sur votre téléphone, ça marche mais cette méthode n'est pas conseillée.

La deuxième façon (conseillé) est de tout mettre dans une StyleSheet. C'est le cas sur le App.js de base. La balise View a l'attribut style={styles.container}. En regardant la StyleSheet (qui s'appelle styles), vous voyez :

```
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
```

Comme en CSS, container correspond à une classe en DevWeb.

Vous pouvez modifier la couleur de fond de votre View :

```
  container: {
```

```
    flex: 1,
    backgroundColor: 'green',
    alignItems: 'center',
    justifyContent: 'center',
  },
```

Le style

Comme en HTML, il y a plusieurs moyens de définir la taille et la position d'un élément. Notez que ce n'est pas uniquement le style de l'élément en lui-même qui en décide : L'élément parent et les éléments fils jouent un rôle important, mais aussi les éléments frères (on l'a vu avec le flex).

Voilà une liste des propriétés que l'on va utiliser :

- `width`, `height`
- `margin`, `marginLeft`, `marginRight`, `marginTop`, `marginBottom`
- `padding`, `paddingLeft`, `paddingRight`, `paddingTop`, `paddingBottom`
- `alignItems`, `justifyContent`
- `position`, `left`, `right`, `top`, `bottom`
- `flex`, `flexDirection`

On va flex

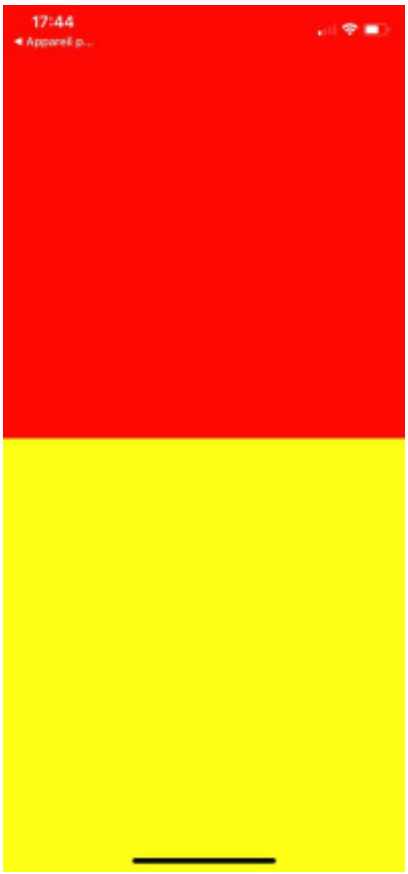
Vous avez dû vous demander à quoi correspond le flex: 1 dans le style de la View.

En développement mobile, l'une des principales difficultés est de s'adapter aux différentes tailles d'écran. Vous développez pour plusieurs tailles de téléphones mais aussi des tablettes. Si l'on impose la taille de façon fixe (par exemple, une hauteur de 300px), ça ne fera pas du tout pareil sur les téléphones.

L'intérêt de flex est de couper l'écrans en plusieurs parties et indiquer quelle part de l'écran chaque élément prendera.

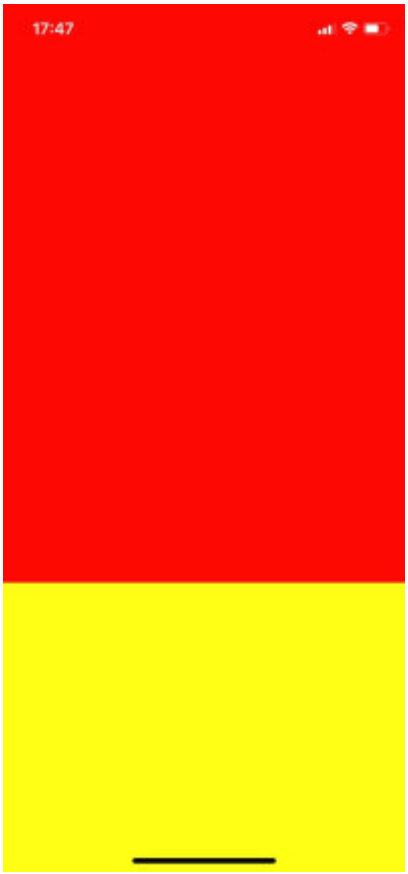
Nous allons faire une exemple :

```
export default function App() {
  return (
    <View style={{flex: 1, backgroundColor: 'green'}}>
      <View style={{flex: 1, backgroundColor: 'red'}}/>
      <View style={{flex: 1, backgroundColor: 'yellow'}}/>
    </View>
  );
}
```



Comme vous le voyez, les deux View dans la View principale prennent chacune une part de l'écran (comme elles sont deux, elles prennent donc chacune une moitié).

Si vous mettez flex: 2 pour la View rouge, vous obtiendrez ceci :



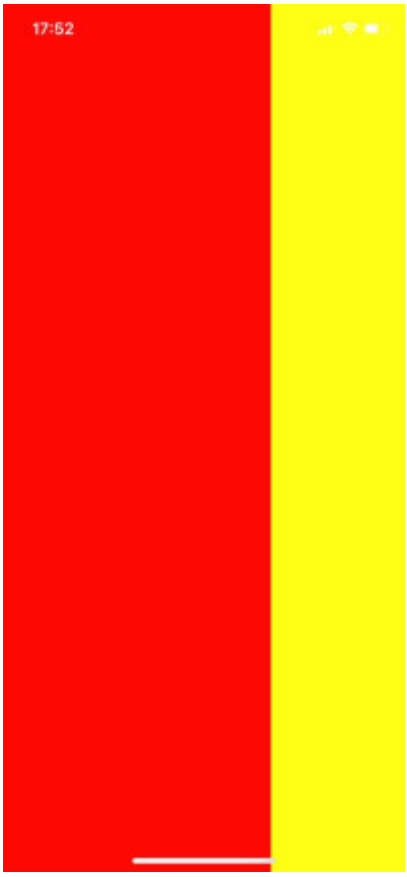
Maintenant, la View rouge prend 2 parts de l'écran et la View jaune une part. La View rouge prend donc deux tiers de l'écran et la View jaune un tiers.

Vous pouvez jouer avec les flex en changeant les proportions ou en ajoutant des View.

Donc dans le code de base du App.js, le flex: 1 indique que la View principale prend tout l'écran (comme elle est toute seule).

On vient de découper des zones horizontales, pour qu'elles deviennent verticales, voilà le code :

```
    <View style={{flex: 1, backgroundColor: 'green', flexDirection: 'row'}}>
      <View style={{flex: 2, backgroundColor: 'red'}}/>
      <View style={{flex: 1, backgroundColor: 'yellow'}}/>
    </View>
```



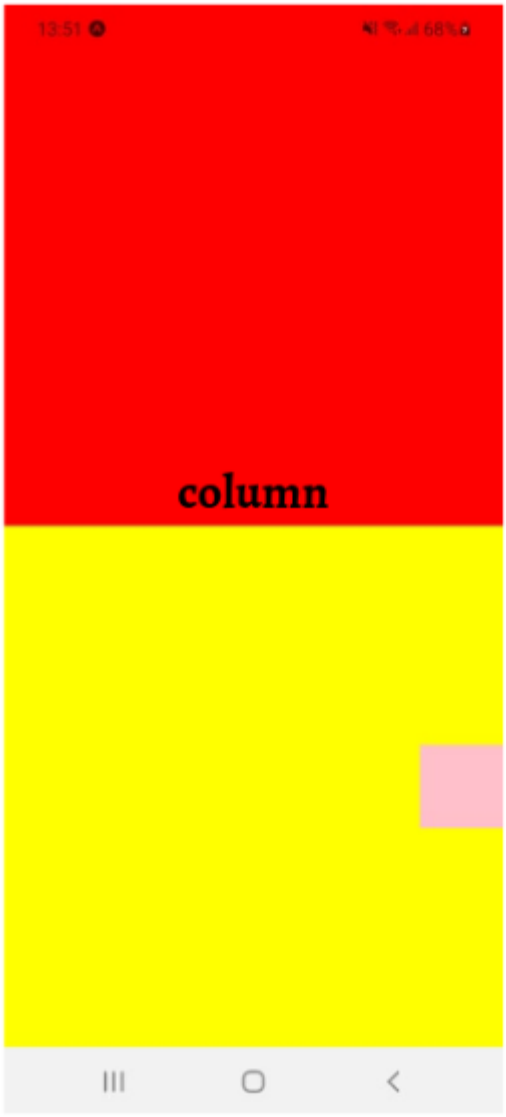
Taille et centrage

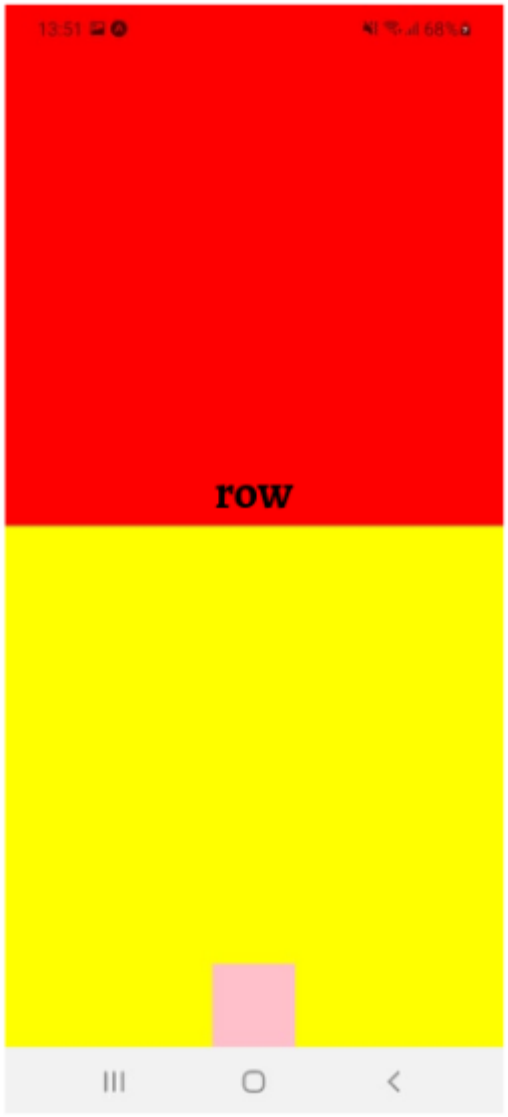
Pour les deux premières, `width` et `height`, je pense que je n'ai pas grand chose à vous expliquer : c'est avec ça que l'on peut donner la largeur et la hauteur de notre élément. On indique la valeur en pixels, c'est un entier.

Pour maîtriser sa position, on va utiliser `alignItems` et `justifyContent`. Ces deux propriétés sont utilisées sur l'élément parent pour déterminer comment son alignés ses fils. On peut leur donner la valeur `flex-end`, `flex-start` ou `center`.

Vous comprenez sûrement que l'une des propriétés sert à l'alignement vertical, l'autre à l'alignement horizontal. Mais qui fait quoi ? C'est là que ça se complique (à peine). Si l'on change la `flexDirection` (on peut mettre `column`, la valeur par défaut, ou `row`), on échange l'effet de `alignItems` et `justifyContent`. En réalité, `justifyContent` fait l'alignement le long de la `flexDirection`, et `alignItems` perpendiculairement.

```
<View style={{flex: 1}}>
    <View style={{flex:1, backgroundColor:'red'}}/>
    <View style={{backgroundColor:'yellow', flex:1, flexDirection:'column',
alignItems:'flex-end', justifyContent: 'center'}}>
        <View style={{backgroundColor:'pink', width:60, height:60}}/>
    </View>
</View>
```





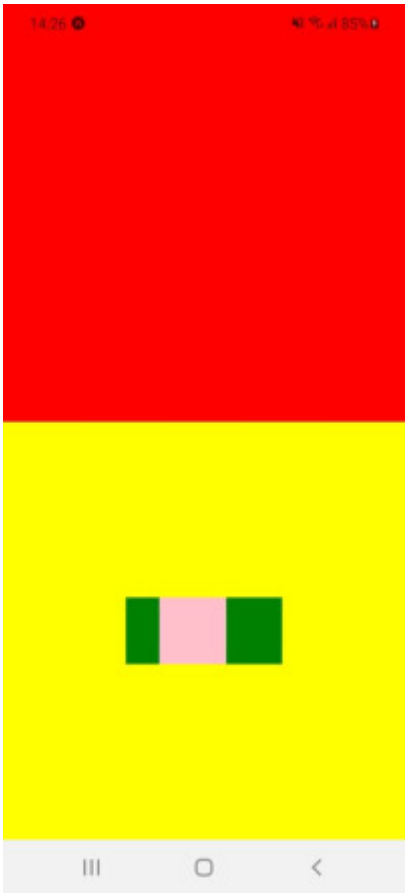
Marges

On peut aussi ajouter des marges à notre élément. Les marges extérieures sont définies par `margin`, les marges intérieures par `padding`. On peut définir toutes les marges d'un coup, on uniquement d'un côté avec les propriétés comme `marginLeft` ou `paddingRight`. On donne simplement un entier, la valeur en pixels.

C'est ici qu'on va voir un concept intéressant : les éléments fils jouent un rôle dans la taille d'un élément. En effet, si on ne précise rien, une `View` ne prends pas plus de place que ce dont son contenu a besoin. On va le voir sur l'exemple suivant : la taille de la `View` verte est uniquement définie par sa marge intérieure à droite (on a décidé de laisser 50 pixels entre le bord droit de la `View` et son contenu); et la marge extérieure à gauche de la `View` rose (on a décidé de laisser 30 pixels entre le bord gauche de la `View` et ce qu'il y a à sa gauche).

```
<View style={{flex: 1}}>
    <View style={{flex:1, backgroundColor:'red'}}/>
    <View style={{backgroundColor:'yellow', flex:1, alignItems:'center',
justifyContent: 'center'}}>
        <View style={{backgroundColor:'green', paddingRight:50}}>
```

```
            <View style={{backgroundColor:'pink', width:60, height:60,
marginLeft: 30}}/>
        </View>
    </View>
</View>
```



Position

Cette partie est plus compliquée, elle ne sera pas abordée durant la séance

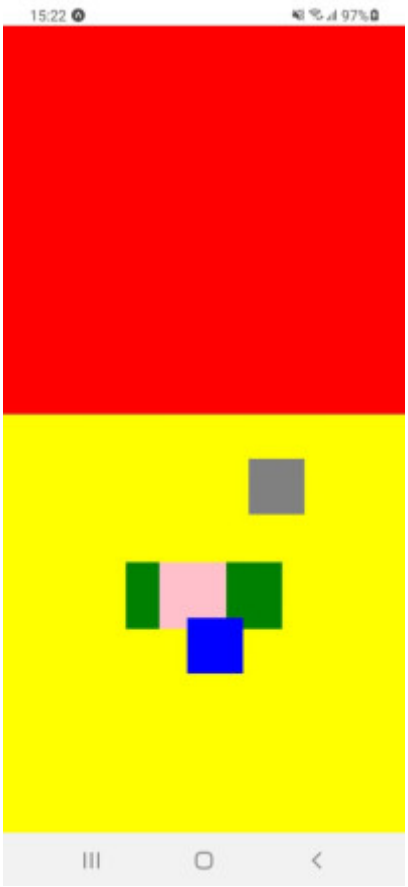
On va maintenant aborder la propriété `position`, qui affecte les effets de `left`, `right`, `top` et `bottom`. Lorsqu'on mets la `position` à `relative` (c'est la valeur par défaut), la position de l'élément est définie avec les propriétés que l'on a vu précédemment. Mais on peut aussi ajouter les propriétés comme `top` ou `right`, pour décaler l'élément de sa position prévue, sans affecter la position des éléments autour. On mets des entiers, la valeur en pixels

Si l'on mets la `position` à `absolute`, la position de l'élément n'est définie que par ces propriétés. On peut donc librement définir l'espace, en pixels, que l'on laisse entre les bords de la `View` parente et les bords de la `View` concernée.

Dans cet exemple, la `View` bleue est décalée par rapport à sa position normale, qui serait centrée en dessous de la `View` rose, on voit qu'elle vient la chevaucher. La `View` grise est en `position: absolute`, on a pu lui donner exactement la position voulue.

```
<View style={{flex: 1}}>
```

```
    <View style={{flex:1, backgroundColor:'red'}}/>
    <View style={{backgroundColor:'yellow', flex:1, alignItems:'center',
justifyContent: 'center'}}>
        <View style={{backgroundColor:'green', paddingRight:50}}>
            <View style={{backgroundColor:'pink', width:60, height:60,
marginLeft: 30}}/>
        </View>
        <View style={{backgroundColor:'blue', width:50, height:50,
bottom:10, left:10}}/>
        <View style={{backgroundColor:'grey', position:'absolute', width:50,
height:50, top:40, right:90}}/>
    </View>
</View>
```



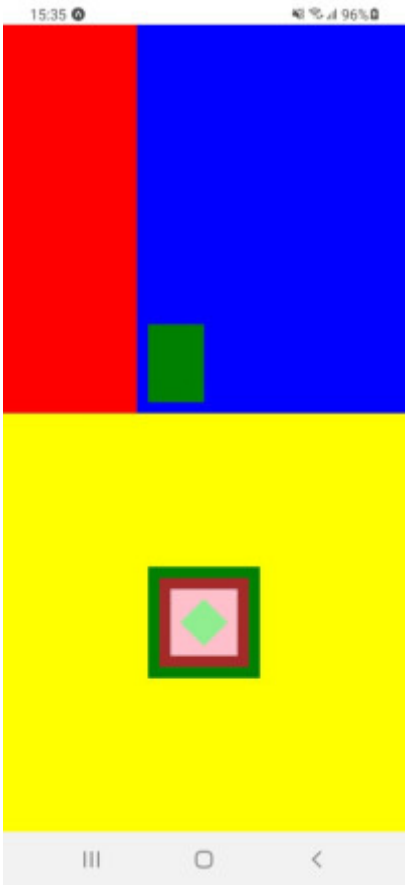
Pratiquez !

Cette partie est plus compliquée, elle ne sera pas abordée durant la séance

Exercice

Essayez de reproduire cela avec les propriétés que nous venons de voir :

La `View` vert clair est plus compliquée, on n'a pas vu la propriété `transform`. Si vous avez du temps, cherchez comment l'utiliser sur la documentation de React Native !



Correction

```
<View style={{flex: 1}}>
  <View style={{flex:1, flexDirection:'row'}}>
    <View style={{flex:1, backgroundColor:'red'}}/>
      <View style={{flex:2, backgroundColor:'blue'}}>
        <View style={{backgroundColor:'green', width:50, height:70,
position:'absolute', bottom:10, left:10}}/>
      </View>
  </View>
  <View style={{backgroundColor:'yellow', flex:1, alignItems:'center',
justifyContent: 'center'}}>
    <View style={{backgroundColor:'green'}}>
      <View style={{backgroundColor:'brown', margin:10, padding:10}}>
        <View style={{backgroundColor:'pink', width:60, height:60,
alignItems:'center', justifyContent: 'center'}}>
          <View style={{backgroundColor:'lightgreen', width:30, height:30,
transform:[{rotate:'45deg'}]}}/>
        </View>
      </View>
    </View>
  </View>
</View>
```

A vous de jouer

Il est temps pour vous de faire votre propre page.

En cas de doute, vous pouvez utiliser la documentation de React Native : https://reactnative.dev/docs/getting-started

Voici quelques éléments :

- Mettre du texte :

```
  <Text> Mon texte </Texte>
```

- Mettre un icône de chargement :

```
  <ActivityIndicator/>
```

- Mettre une image :

```
  <Image source={{uri : 'https://img.20mn.fr/vmusmUDoT_enbz8ZVHKhGA/830x532_loutre-cendree.jpg'}}/>
```

- Mettre un lien (ne pas oublier import { Linking } from 'react-native') :

```
  <Text onPress={() => Linking.openURL('http://google.com')}> Google </Text>
```

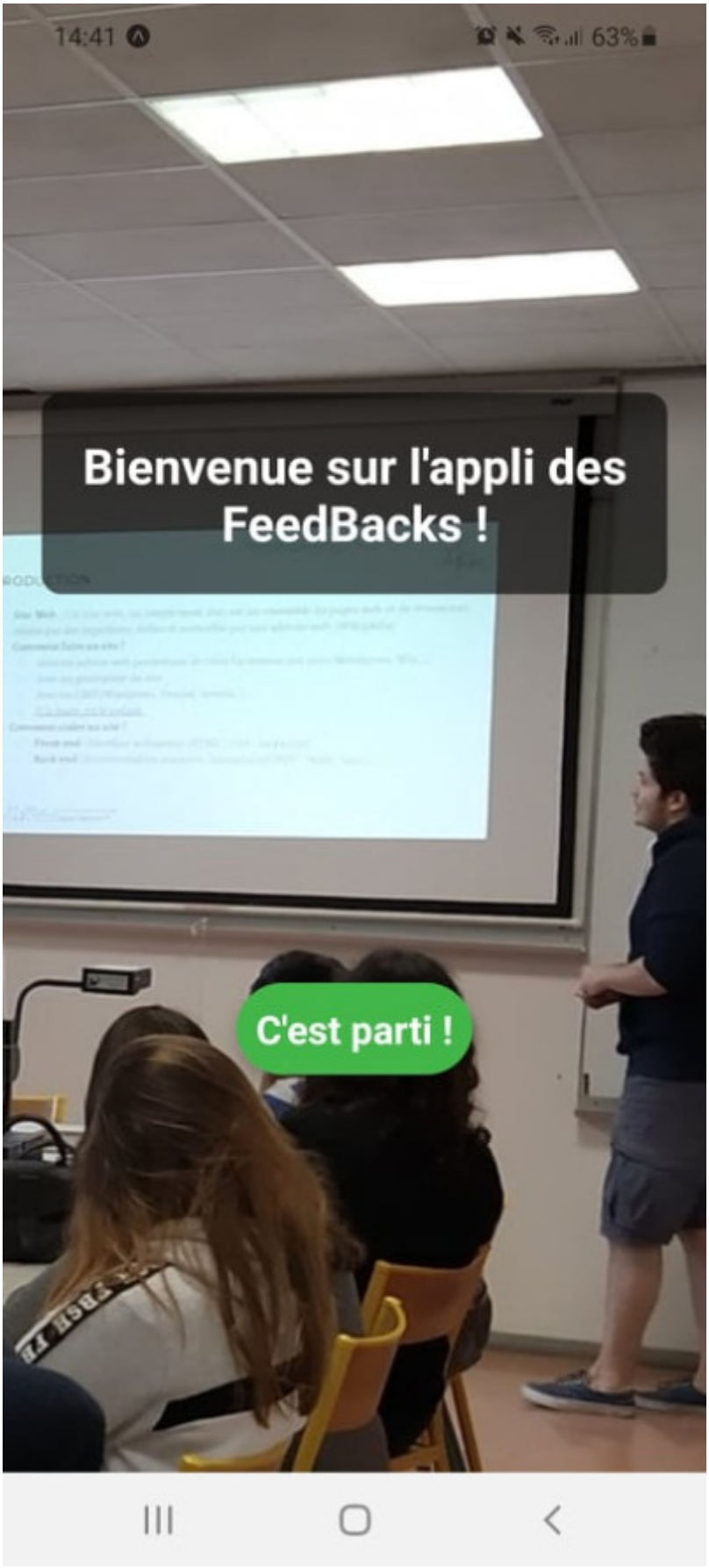
Si vous n'avez pas d'inspiration, voici un exemple de page que vous pouvez essayer de reproduire :

Un peu d'aide :

On doit séparer notre images en plusieurs `View`, et choisir le bon `flex` pour chacune. Voilà le découpage :

Un peu d'avance sur la suite

Dans les prochaines formations, nous allons essayer de réaliser une application utile au GInfo. Elle permettra aux utilisateurs de laisser anonymement des feedbacks sur les formations. On peut commencer par réaliser la page d'acceuil :



L'image de fond :