

Chapitre 6 : Introduction à Flutter

Table des matières

6. Chapitre 6 : Introduction à Flutter	4
6.1. Introduction	4
6.2. Architecture de Flutter	4
6.3. Fonctionnement	5
6.4. Les Widgets	6
6.5. Introduction à Dart	7
6.5.1. Environnements de Programmation Dart	7
6.5.2. Syntaxe	7
6.5.3. Les types	8
6.5.4. La boucle pour	9
6.5.5. Les listes	9
6.5.6. Les Maps	10
6.5.7. La fonctions Lambda	11
6.5.8. Les classes	11
6.5.9. Quelques bibliothèques utiles	12
6.6. Catalogue des Widgets de Flutter	12
6.6.1. Les Widgets Flutter sans état	13
6.6.2. Les Widgets Flutter avec état	13
6.6.3. La classe IconButton	13
6.6.4. La classe FloatingActionButton	13
6.6.5. La classe Text	14
6.6.6. La classe TextField	14
6.6.7. La classe Image	15
6.6.8. la classe Boutton Radio	15

6.6.9. La classe Checkbox	16
6.7. Conclusion	16
Bibliographie et Webographie	18

6. Chapitre 6 : Introduction à Flutter

6.1. Introduction

Flutter est une boîte à outils d'interface utilisateur conçue pour permettre la réutilisation du code sur des systèmes d'exploitation mobiles différents comme iOS et Android. Cela en générant un code natif multiplateforme afin de permettre aux applications d'interagir directement avec les services de plateforme sous-jacents. L'objectif est de permettre aux développeurs de fournir des applications hautes performances natives sur différentes plateformes, en tenant compte des différences tout en partageant le maximum de code.

Pendant le développement, les applications Flutter s'exécutent dans une machine virtuelle qui offre un rechargement à chaud avec état des modifications sans avoir besoin d'une recompilation complète. Cependant, les applications Flutter sont compilées directement en code machine, qu'il s'agisse d'instructions Intel x64 ou ARM, ou en JavaScript si elles ciblent le Web. Le framework est open source, avec une licence BSD permissive, et dispose d'un écosystème florissant de packages tiers qui complètent les fonctionnalités de base.

6.2. Architecture de Flutter

Flutter est conçu comme un système extensible en couches. Il existe sous la forme d'une série de bibliothèques indépendantes qui dépendent chacune de la couche sous-jacente. Aucune couche n'a un accès privilégié à la couche inférieure, et chaque partie du niveau du framework est conçue pour être facultative et remplaçable [20].

Pour le système d'exploitation sous-jacent, les applications Flutter sont packagées de la même manière que toute autre application native. Un embedder spécifique à la plate-forme fournit un point d'entrée et se coordonne avec le système d'exploitation sous-jacent pour l'accès aux services tels que les rendus visuels, l'accessibilité et l'entrée, et gère les messages d'événements. L'embedder est écrit dans un langage adapté à la plateforme native sur laquelle il est exécuté: actuellement Java et C++ pour Android, Objective-C/Objective-C++ pour iOS et macOS, et C++ pour Windows et Linux. En utilisant l'embedder, le code Flutter peut être intégré dans une application existante en tant que module, ou le code peut être l'intégralité du contenu de l'application [20].

Au cœur de Flutter se trouve le moteur Flutter, qui est principalement écrit en C++ et prend en charge les primitives nécessaires pour toutes les applications Flutter. Le moteur fournit l'implémentation de bas niveau de l'API principale de Flutter, y compris les graphiques (via Skia [21]), la disposition du texte, les E/S de fichiers et de réseau, la prise en charge de l'accessibilité, l'architecture de plug-in et une chaîne d'outils d'exécution et de compilation Dart.

Flutter est utilisable via un framework moderne et réactif écrit avec le langage Dart. Il comprend un ensemble riche de bibliothèques de mise en page et de base, composé d'une série de couches qui sont [20]:

- 1- Des classes fondamentales de base et des services tels que l'animation, [la peinture](#) [et les gestes](#) qui offrent des abstractions couramment utilisées sur la base sous-jacente.
- 2- La couche de rendu fournit une abstraction de classes pour traiter la mise en page. Utilisée pour créer une arborescence d'objets manipulable de manière dynamique.
- 3- La couche widgets est une abstraction de composition. Chaque objet de rendu dans la couche de rendu a une classe correspondante dans la couche de widgets. De plus, la couche widgets permet de définir des combinaisons de classes. C'est la couche à laquelle le modèle de programmation réactif est introduit.
- 4- Les bibliothèques Material et Cupertino offrent des ensembles complets de contrôles qui utilisent les primitives de composition de la couche de widgets.

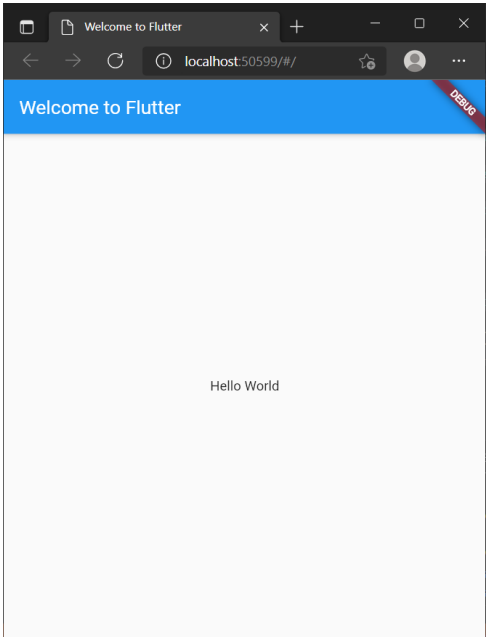
6.3. Fonctionnement

L'utilisateur crée la description de l'interface graphique et le framework s'occupe d'utiliser cette configuration pour créer et/ou mettre à jour l'interface utilisateur selon les besoins. Dans Flutter, les widgets sont représentés par des classes immuables qui sont utilisées pour configurer une arborescence d'objets. Flutter est, à la base, une série de mécanismes pour parcourir efficacement les parties modifiées des arbres, et propager les changements à travers ces arbres. Un widget déclare son interface utilisateur en remplaçant la méthode build(). Cette méthode est par conception rapide à exécuter, ce qui lui permet d'être appelée par le framework chaque fois que nécessaire.

6.4. Les Widgets

Flutter met l'accent sur les widgets en tant qu'unité de composition. Les widgets sont les éléments constitutifs de l'interface utilisateur d'une application Flutter, et chaque widget est une déclaration immuable d'une partie de l'interface utilisateur.

Les widgets forment une hiérarchie basée sur la composition. Chacune d'elle est imbriquée dans son parent. Cette structure s'étend jusqu'au widget racine (le conteneur qui héberge l'application Flutter). Ci-dessous un code basique de l'application hello Word :

<pre>lib > main.dart > ... 1 import 'package:flutter/material.dart'; 2 Run Debug Profile 3 void main() => runApp(const MyApp()); 4 5 class MyApp extends StatelessWidget { 6 const MyApp({Key? key}) : super(key: key); 7 8 @override 9 Widget build(BuildContext context) { 10 return MaterialApp(11 title: 'Welcome to Flutter', 12 home: Scaffold(13 appBar: AppBar(14 title: const Text('Welcome to Flutter'), 15), // AppBar 16 body: const Center(17 child: Text('Hello World'), 18), // Center 19), // Scaffold 20); // MaterialApp 21 } 22 }</pre>	
Code source	Résultat

L'application étend StatelessWidget, ce qui fait de l'application elle-même un widget. Dans Flutter, presque tout est un widget, y compris l'alignement, le remplissage et la mise en page. Le widget Scaffold, de la bibliothèque de matériaux, fournit une barre d'application par défaut et une propriété de corps qui contient l'arborescence des widgets pour l'écran d'accueil.

Le travail principal d'un widget est de fournir une méthode build() qui décrit comment afficher le widget en termes d'autres widgets de niveau inférieur.

Le corps de cet exemple se compose d'un widget Center contenant un widget enfant Text. Le widget Centre aligne sa sous-arborescence de widgets au centre de l'écran.

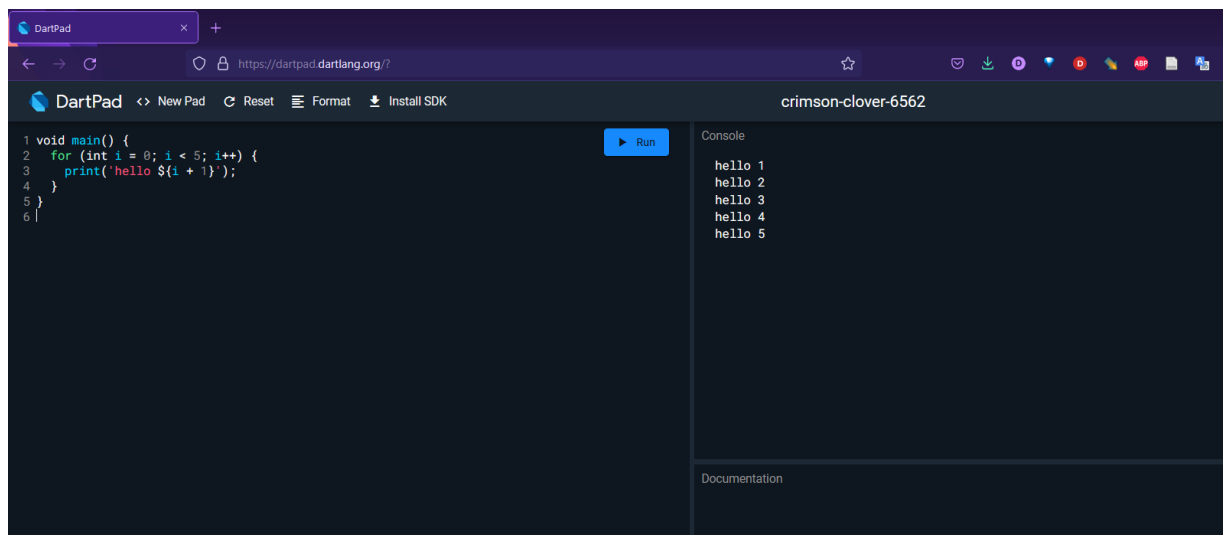
6.5. Introduction à Dart

Dart est un langage orienté objet avec une syntaxe de style C qui peut éventuellement transcompiler en JavaScript. Il prend en charge une gamme variée d'aides à la programmation comme les interfaces, les classes, les collections, les génériques et la saisie facultative.

Dart peut être largement utilisé pour créer des applications d'une seule page. Les applications d'une seule page s'appliquent uniquement aux sites Web et aux applications Web. Les applications à page unique permettent de naviguer entre les différents écrans du site Web sans charger une page Web différente dans le navigateur. Un exemple classique est Gmail, lorsque un clique est effectué sur un message dans la boîte de réception, le navigateur reste sur la même page Web, mais le code JavaScript masque la boîte de réception et affiche le corps du message à l'écran.

6.5.1. Environnements de Programmation Dart

Un ensemble d'IDE supporte l'installation de Dart en tant que plugin comme Eclipse, IntelliJ, VS Code, et WebStorm. De plus, Il existe un éditeur en ligne appelé DartPad accessible sur <https://dartpad.dartlang.org/>. L'éditeur Dart exécute le script et affiche à la fois le HTML ainsi que la sortie de la console.



6.5.2. Syntaxe

La syntaxe définit un ensemble de règles pour l'écriture de programmes. Chaque spécification de langage définit sa propre syntaxe. Un programme Dart est composé de :

- Variables et opérateurs
- Des classes
- Les fonctions
- Expressions et constructions de programmation
- Prise de décision et constructions en boucle
- commentaires
- Bibliothèques et packages
- Définition des types

Ci-dessous, un exemple 'Hello Word' écrit avec Dart

```
void main() {  
  print('Hello Word');  
}
```

La fonction `main()` est une méthode prédéfinie dans Dart. Cette méthode sert de point d'entrée à l'application. Un script Dart a besoin de la méthode `main()` pour s'exécuter. `print()` est une fonction prédéfinie qui imprime la chaîne ou la valeur spécifiée sur la sortie standard, c'est-à-dire le terminal.

6.5.3. Les types

L'une des caractéristiques les plus fondamentales d'un langage de programmation est l'ensemble des types de données qu'il prend en charge. Ce sont le type de valeurs qui peuvent être représentées et manipulées dans un langage de programmation.

Le langage Dart prend en charge les types suivants :

- Nombres
- Chaîne de caractère
- Booléens
- Listes
- Map

Dart est un langage à typage optionnel. Si le type d'une variable n'est pas explicitement spécifié, le type de la variable est dynamique. Le mot-clé *dynamic* peut également être utilisé explicitement comme annotation de type.

6.5.4. La boucle pour

La boucle ‘pour’ est utilisée pour les itérations, le code ci-dessous montre un exemple de son utilisation.

```
void boucles() {  
  debut:  
  for (int i = 1; i <= 10; i++) {  
    print('i= ${i}');  
    if (i >= 5) {  
      continue debut;  
    }  
    break debut;  
  }  
}
```

6.5.5. Les listes

La collection la plus courante dans presque tous les langages de programmation est peut-être le tableau ou le groupe ordonné d'objets. Dans Dart, les tableaux sont des objets List. La déclaration des listes peut être faite de plusieurs manières comme montré ci-dessous :

```
List list = List.empty(growable: true);  
List<String> list2 = List.filled(5, '0', growable: true);  
List list3 = [];
```

Dans la déclaration de la variable ‘list’ dans la première ligne du code précédent, Si [growable] est false, la liste aura une longueur fixe égale à 0. Si [growable] est vrai, la liste est évolutive.

La deuxième ligne du code précédent montre la déclaration d’une liste avec 5 éléments de type chaîne de caractères initialisé.

La troisième déclaration montre une création d’une liste évolutive non initialisée.

L’ajout d’un élément à une liste se fait par la méthode *add* comme montré par le code ci-dessous.

```
for (int i = 1; i <= 10; i++) {  
  list.add(i);  
}
```

Cependant, le parcours des listes peut être fait en utilisant la méthode *forEach* comme montré par le code ci-dessous.

```
list.forEach((element) {  
    print(element);  
});
```

Ou tout simplement avec la fonction Lambda comme suit :

```
list.forEach((element) => print(element));
```

De plus, l'objet 'Set' peut être utilisé comme une liste non triée et d'items uniques. Elle est utilisée de la même manière qu'une liste. Le code ci-dessous montre comment la déclarer et comment la parcourir.

```
// Déclarer la variable Set1 de type Set  
Set set1 = <String>{};  
  
// Initialiser Set1  
set1 = {'mot1', 'mot2', 'mot3'};  
  
// Déclarer une variable de type Set  
//qui prend ses valeurs de la variable List  
Set set2 = Set.from(list);  
  
// Ajouter des éléments à la liste Set2  
for (int i = 1; i <= 10; i++) {  
    set2.add(i);  
}  
  
// Parcourir les éléments de Set2  
set2.forEach((element) => print(element));
```

6.5.6. Les Maps

En général, une Map est un objet qui associe des clés et des valeurs. Les clés et les valeurs peuvent être de n'importe quel type d'objet. Chaque clé n'apparaît qu'une seule fois, contrairement aux valeurs. Ci-dessous un exemple de déclaration de variable de type Map :

```
Map mVoiture = {  
    'marque': 'seat',  
    'model': 'Ibiza',  
    'Couleur': 'bleu',  
    'année': 2015  
};
```

Le type Map offre la possibilité de ramener des objets d'une autre variable de type Map avec la méthode addAll.

```
map2.addAll(mVoiture);
```

Pour Ajouter une clé valeur ou pour en modifier une, on peut accéder via les indexes comme montré ci-dessous :

```
mVoiture['propriétaire'] = 'valeur';
```

Cependant, le parcours des éléments d'une map peut être fait pour chaque clé valeur comme suit :

```
mVoiture.forEach((key, value) {  
    print(key + '=' + value.toString());  
});
```

Comme on peut récupérer la liste des clés ou des valeurs et les parcourir comme montré ci-dessous :

```
var keys = mVoiture.keys;  
for (var k in keys) print (k);
```

6.5.7. La fonctions Lambda

Les fonctions lambda sont un mécanisme concis pour représenter des fonctions.

Méthodes sans la fonction Lambda	Méthodes écrite avec la fonction Lambda
<pre>void printMessage({message}) { print(message); }</pre>	<pre>printMessage({message})=>print(message);</pre>
<pre>int produit(x, y) { return x * y; }</pre>	<pre>int produit(x, y) => x * y;</pre>

6.5.8. Les classes

Dart est un langage orienté objet qui adopte la notion des classes et d'héritage. Chaque objet est une instance d'une classe, et toutes les classes sauf Null descendent de la classe 'Object'. L'héritage basé sur un mélange 'Mixin' signifie que chaque classe ait une superclasse et un corps qui peut être réutilisé à plusieurs niveau de hiérarchie.

Ci-dessous un exemple de création de classe voiture, elle contient des attributs de type non initialisés, un constructeur, et une méthode qui affiche les propriétés du véhicule.

```
class Voiture {  
    var marque;  
    var model;  
    var couleur;  
    var annee;
```

```

Voiture({marque, model, couleur, annee}) {
  this.marque = marque;
  this.model = model;
  this.couleur = couleur;
  this.annee = annee; }
void afficherVoiture() {
  print('marque : ${marque} ,' +
    'model : ${model},' +
    'couleur : ${couleur},' +
    'année : ${annee}');
}
}

```

6.5.9. Quelques bibliothèques utiles

Une bibliothèque représente un ensemble d'instructions de programmation. Dart possède des bibliothèques intégrées qui peuvent être utilisées pour stocker des routines. Les bibliothèques Dart incluent des collections de classes, de constantes, de fonctions, de définitions de type, de propriétés et d'exceptions [22]. Ci-dessous les bibliothèques Dart les plus utilisées :

- dart:io
 - utilisée pour prendre en charge les entrées/sorties comme les fichiers, sockets, et le HTTP. Cette bibliothèque est importée par défaut et ne nécessite pas de l'ajouter à l'entête du programme.
- dart:core
 - Cette bibliothèque offre les différents types de données et des fonctionnalités de base intégrés. De même, elle est importée par défaut.
- dart: math
 - Fournit des constantes et des fonctions mathématiques.
- dart: convert
 - Cette bibliothèque offre les fonctionnalités d'encodeurs et de décodeurs pour la conversion entre les données, y compris JSON et UTF-8.

6.6. Catalogue des Widgets de Flutter

Le site officiel de Flutter [23] offre la liste de toutes les Widgets, leurs fonctionnalités, ainsi que la documentation technique qui facilite l'intégration de ces Widgets dans les applications. Toutes les propriétés qui contiennent les Widgets sont définitives et ne peuvent être définies que lorsque le widget est initialisé. Cela les maintient légers et faciles à recréer lorsque

l'arborescence des widgets est rechargée. Il existe deux types de widgets : sans état et avec état.

6.6.1. Les Widgets Flutter sans état

Les widgets sans état sont des widgets qui ne stockent pas d'état qui sont en général des Valeurs de variable et des états d'objets. Par exemple, une icône est sans état où l'image de l'icône est définie une fois pour tous lorsque de sa création. De même, un widget Texte est également sans état, Si sa valeur est modifiée au cours d'exécution du programme, il sera recréer comme un nouveau widget avec le nouveau texte.

6.6.2. Les Widgets Flutter avec état

Un widget avec état signifie qu'il peut suivre les modifications et mettre à jour l'interface utilisateur en fonction de ces changement. Le widget avec état lui-même est immuable, mais il crée un objet State qui garde une trace des modifications effectuées. Lorsque les valeurs de l'objet State changent, il crée un tout nouveau widget avec les valeurs mises à jour. Ainsi, le widget léger est recréé mais l'état persiste à travers les changements [24].

6.6.3. La classe IconButton

Un bouton d'icône est une image imprimée qui réagit au toucher. Les boutons d'icônes sont couramment utilisés dans le champ AppBar.actions, mais ils peuvent également être utilisés à de nombreux autres endroits. Ci-dessous un exemple de déclaration d'un bouton icône :

```
IconButton(  
  icon: const Icon(Icons.android),  
  color: Colors.white,  
  onPressed: () {},  
),
```

6.6.4. La classe FloatingActionButton

Un bouton d'action flottant est un bouton d'icône circulaire qui survole le contenu pour promouvoir une action dans l'application.

```
floatingActionButton: FloatingActionButton(  
  onPressed: () {  
    // Add your onPressed code here!  
  },
```

```

        backgroundColor: Colors.green,
        child: const Icon(Icons.navigation),
      ),

```

6.6.5. La classe Text

Le widget Texte affiche une chaîne de texte avec un style unique. La chaîne peut s'étendre sur plusieurs lignes ou s'afficher toutes sur la même ligne en fonction des contraintes de mise en page.

```

Text(
  'Salut',
  textAlign: TextAlign.center,
  overflow: TextOverflow.ellipsis,
  style: const TextStyle(fontWeight: FontWeight.bold),
)

```

6.6.6. La classe TextField

Un champ TextField permet à l'utilisateur de saisir du texte, soit avec un clavier matériel, soit avec un clavier à l'écran.

Le champ de texte appelle la méthode onChanged chaque fois que l'utilisateur modifie le texte. Si l'utilisateur indique qu'il a fini de taper dans le champ (par exemple, en appuyant sur un bouton du clavier logiciel), le champ de texte appelle onSubmit. Ci-dessous le code source d'un TextField une fois rempli et le bouton ok appuyé, on affiche un message à l'écran.

```

TextField(
  onSubmit: (String value) async {
    await showDialog<void>(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: const Text('Text Saisi'),
          content: Text(
            "$value"),
          actions: <Widget>[
            TextButton(
              onPressed: () {
                Navigator.pop(context);
              },
              child: const Text('OK'),
            ),
          ],
        );
      },
    );
  },
)

```

```

    },
  );
},
),

```

6.6.7. La classe Image

Plusieurs constructeurs sont fournis pour les différentes façons dont une image peut être spécifiée :

- Image, pour obtenir une image à partir d'un ImageProvider.
- Image.asset, pour obtenir une image d'un AssetBundle.
- Image.network, pour obtenir une image à partir d'une URL.
- Image.file, pour obtenir une image à partir d'un fichier.
- Image.memory, pour obtenir une image à partir d'une Uint8List.

Exemple d'utilisation :

```

const Image(
  image: NetworkImage('https://flutter.github.io/assets-for-api-
docs/assets/widgets/owl.jpg'),
)

Image.network('https://flutter.github.io/assets-for-api-
docs/assets/widgets/owl-2.jpg')

image: AssetImage('assets/google_play_100.png')

```

6.6.8. la classe Boutton Radio

les boutons radio sont utilisés pour choisir entre des valeurs mutuellement exclusives. Le bouton radio lui-même ne conserve aucun état. Au lieu de cela, la sélection de la radio invoque la méthode onChanged. Ci-dessous un exemple d'utilisation des Bouttons radio :

```

Column(
  children: <Widget>[
    ListTile(
      title: const Text('valeur1'),
      leading: Radio<v1>(
        value: v1.valeur1,
        groupValue: groupe,
        onChanged: (v1? value) {

```

```

        setState(() {
          groupe = value;
        });
      },
    ),
  ),
  ListTile(
    title: const Text('valeur2'),
    leading: Radio<V1>(
      value: V1.valeur2,
      groupValue: groupe,
      onChanged: (V1? value) {
        setState(() {
          groupe = value;
        });
      },
    ),
  ),
],
);

```

6.6.9. La classe Checkbox

La classe case à cocher ne conserve pas d'état. Dans le cas de modification de sa valeur, elle appelle la méthode `onChanged` pour réaliser une opération. Le code ci-dessous montre un exemple d'utilisation des cases à cocher :

```

Checkbox(
  checkColor: Colors.white,
  fillColor: MaterialStateProperty.resolveWith(getColor),
  value: isChecked,
  onChanged: (bool? value) {
    setState(() {
      isChecked = value!;
    });
  },
);

```

6.7. Conclusion

Dans ce chapitre, on a présenté les éléments de base du langage Dart et qui sont les plus rencontrés avec le Framework Flutter. A la fin de ce chapitre combiné avec son TP, l'étudiant va pouvoir installer et configurer l'environnement de développement Flutter à base de Visual Studio Code et développer des applications simples avec Flutter. L'étudiant peut aussi comprendre les codes sources autogénérés, d'importer et d'intégrer du code de différents

Widgets fournit par Google. Cela permettra à l'étudiant d'aller plus loin avec une autoformation sur des centaines de Widgets qui lui seront indispensables.

Bibliographie et Webographie

- [20] « Aperçu du cadre que chaque développeur de flutter devrait connaître ». <https://cdmana.com/2022/01/202201191925573961.html> (consulté le 1 février 2022).
- [21] « Skia », *Skia*. <https://skia.org/> (consulté le 1 février 2022).
- [22] « Dart-programming-libraries — Get Docs ». <https://getdoc.wiki/Dart-programming-libraries> (consulté le 2 février 2022).
- [23] « Flutter widget index ». <https://docs.flutter.dev/reference/widgets> (consulté le 2 février 2022).
- [24] « Flutter widgets: Their different types and how to create them », *Software outsourcing company*, 6 août 2020. <https://www.arrowwhitech.com/flutter-widgets-their-different-types-and-how-to-create-them/> (consulté le 3 février 2022).
- [25] « Développement d'une application native Android », https://elearning-facsci.univ-annaba.dz/pluginfile.php/33732/mod_resource/content/1/TP1-%20D%C3%A9veloppement%20d%E2%80%99une%20application%20native%20Android%20-%20cr%C3%A9ation%20de%20maquette%20et%20un%20nouveau%20projet%20avec%20Visual%20Studio.pdf (consulté le 3 février 2022).
- [26] « Développement d'une application native Android (Suite) », https://elearning-facsci.univ-annaba.dz/pluginfile.php/33733/mod_resource/content/1/TP2-%20cr%C3%A9ation%20de%20ressources.pdf (consulté le 3 février 2022).