INFORMATIQUE
TECHNIQUE

Expert IT

Flutter™

Développez vos applications mobiles multiplateformes avec Dart

En téléchargement

code des exemples et démonstrations du livre

+ QUIZ

Version en ligne **OFFERTE !** pendant 1 an

Julien TRILLARD

eni

Expert IT

Flutter™

Développez vos applications mobiles multiplateformes avec Dart

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **EIFLUT** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

1. Pourquoi cet ouvrage ? 11
   1.1 Comment faire son choix ? 12
   1.2 Et dans l'univers du développement mobile ? 13
2. À qui s'adresse cet ouvrage ? 13
3. Structure de l'ouvrage 14

Chapitre 1
Introduction

1. État des lieux du développement mobile 15
   1.1 Un marché en progression 15
   1.2 Application Android 16
   1.3 iOS 16
   1.4 Le multiplateforme 17
2. Flutter 18
   2.1 Présentation 18
   2.2 Historique et versions 19
   2.3 Dart 20
   2.4 Développement cross-platform 21
   2.5 Le hot reload 21
   2.6 Organisation par widgets 22

Chapitre 2
Installation de l'environnement

1. Introduction 25
2. Installation de Flutter sur Windows 26
   2.1 Compatibilité 26
   2.2 Téléchargement du SDK Flutter 26
   2.3 Installation du SDK 27
3. Installation de Flutter sur macOS 31
   3.1 Compatibilité 31
   3.2 Téléchargement du SDK Flutter 32
   3.3 Installation du SDK 32
      3.3.1 Mojave (et les versions précédentes) 35
      3.3.2 Catalina 37
   3.4 Installation de cocoapods 42
   3.5 Installation de Xcode 42
4. Installation de Flutter sur Linux 44
   4.1 Compatibilité 44
   4.2 Téléchargement du SDK Flutter 45
   4.3 Installation du SDK 45
5. Mise à jour du SDK 48
   5.1 Connaître la version de Flutter installée 48
   5.2 Être au courant des nouveautés et des mises à jour 49
   5.3 Faire la mise à jour du SDK 50
6. Installation d'Android Studio 50
   6.1 Introduction 50
   6.2 Pourquoi choisir Android Studio ? 50
   6.3 Installation d'Android Studio 51
   6.4 Installation des plug-ins Flutter et Dart 53
7. Flutter Doctor 55
   7.1 Introduction 55
   7.2 Analyse de Flutter Doctor 55

7.3 Corrections et validations 56
7.3.1 Licence Android 56
7.3.2 Plug-ins et autres IDE (Visual Studio Code ou IntelliJ) .. 57
7.3.3 Émulateurs 57

Chapitre 3
Ma première application

1. Introduction 59
2. Création d'un nouveau projet Flutter 60
2.1 Choix du type de projet 61
2.2 Configuration de l'application 62
3. Le contenu de l'application 65
3.1 L'architecture de l'application 65
3.2 Le code de l'application 69
4. Les émulateurs 74
4.1 Introduction 74
4.2 Paramétrage de l'émulateur Android 75
4.3 Lancement de la première application 83
5. Premiers widgets 86

Chapitre 4
Dart, les fondamentaux

1. Introduction 97
2. Syntaxe de base 99
2.1 Variables 99
2.1.1 Les nombres 99
2.1.2 Les chaînes de caractères 100
2.1.3 Les booléens 102
2.1.4 Types dynamiques 102
2.2 Constantes 104

2.3 Collections 105
2.3.1 Listes 105
2.3.2 Maps 107
2.4 Alternatives 108
2.4.1 If 108
2.4.2 Switch 109
2.4.3 Opérateur ternaire 109
2.4.4 Opérateur de nullité 109
2.5 Boucles 110
2.5.1 For 110
2.5.2 Foreach 111
2.5.3 While 111
2.5.4 Do While 112

Chapitre 5
Dart, notions avancées

1. Introduction 113
2. Fonctions 114
2.1 Introduction 114
2.2 Fonction simple 114
2.3 Fonctions avec retour 115
2.4 Fonctions avec paramètres 116
2.4.1 Forme classique 116
2.4.2 Paramètres optionnels 117
2.5 Fonctions anonymes 118
2.6 Fonctions fléchées ou fonctions lambda 119
3. Programmation orientée objet (POO) 120
3.1 Introduction 120
3.2 Classe 120
3.3 Attributs 121
3.4 Constructeur 122
3.5 Instanciation 124

3.6 Accesseurs et mutateurs (getters et setters) 126
3.7 Association 129
3.8 Héritage 133
3.9 Surcharge de méthodes 136
3.10 Interface 138
3.11 Mixin 140
3.12 Extension 142
3.13 Static 143

4. Généricité 145

5. Exception 150
5.1 Exceptions systèmes 151
5.2 Exceptions personnalisées 155
5.3 Exceptions multiples 158

6. Traitement asynchrone 159

Chapitre 6
Les widgets de base

1. Introduction 161

2. Material Component 162
2.1 Scaffold 162
2.1.1 AppBar 162
2.2 floatingActionButton 166
2.3 FloatingActionButtonLocation 168
2.4 bottomNavigationBar 170
2.4.1 BottomAppBar 170
2.4.2 BottomNavigationBar 172
2.5 Drawer 175

3. Cupertino (Style iOS) 178
3.1 Scaffold de base 179
3.2 Scaffold avec barre de navigation 180
3.3 CupertinoNavigationBar 183

4. Détection de la plateforme 185
5. Boutons 193
5.1 FloatingActionButton 193
5.2 RaisedButton 193
5.3 FlatButton 197
5.4 IconButton 199
5.5 OutlineButton 204
5.6 CupertinoButton 206
6. Médias 209
6.1 Icônes 209
6.1.1 Icons (Material Design) 210
6.1.2 CupertinoIcons (Style iOS) 212
6.1.3 FontAwesome 214
6.2 Images 218
6.2.1 Images du projet 218
6.2.2 Images du Web 221
6.3 Texte 223
6.3.1 Text 223
6.3.2 Mise en forme du texte 223
6.3.3 Choix de la police 227
6.3.4 Google Fonts 229
6.4 Layouts (mise en page) 231
6.4.1 Center 231
6.4.2 Column 231
6.4.3 Row 236
6.4.4 Expanded 237
6.4.5 Container 239
6.4.6 Padding 247
6.4.7 Divider 249
6.4.8 Stepper 250

Chapitre 7
Les états

1. Introduction 255
2. StatelessWidget 256
3. StatefulWidget 258
   3.1 Cycle de vie 262
4. Couplage interface et logique 264
   4.1 Provider 265
   4.2 BloC 272

Chapitre 8
Pop-up et navigation

1. Introduction 279
2. SnackBar 280
3. AlertDialog et CupertinoAlertDialog 285
   3.1 AlertDialog 286
   3.2 CupertinoAlertDialog 289
4. SimpleDialog 290
   4.1 SimpleDialog 291
   4.2 CupertinoDialog 300
5. Feuilles de choix 301
   5.1 BottomSheet 301
   5.2 CupertinoActionSheet 306
6. Routes 310
   6.1 Navigator.push 311
   6.2 Navigator.pop 313
   6.3 Navigator.pushNamed 314

Chapitre 9
Formulaires

1. Introduction 321
2. Les champs de saisie de texte 321
   2.1 TextField 322
   2.2 CupertinoTextField 330
   2.3 Types de données 333
      2.3.1 Mot de passe 333
      2.3.2 Autres données 333
   2.4 TextEditingController 334
3. Dates et heures 337
   3.1 Material Design 337
      3.1.1 Date 337
      3.1.2 Heure 344
   3.2 Cupertino 347
      3.2.1 Date 347
      3.2.2 Heure 351
4. Switch 355
   4.1 Material Design 355
   4.2 Cupertino 358
5. Radio 359
6. Checkbox 362
7. Slider 366
   7.1 Material Design 366
   7.2 Cupertino 368

Chapitre 10
Listes et grilles

1. Introduction 371
2. ListView 372
   2.1 ListView.builder 372
   2.2 Dismissible 375
   2.3 ListTile 377
   2.4 ListView.separated 381
   2.5 Personnalisation 383
      2.5.1 Card 383
      2.5.2 Inkwell 385
3. GridView 391
   3.1 GridView.builder 391
   3.2 GridTileBar 392
   3.3 Personnalisation 393

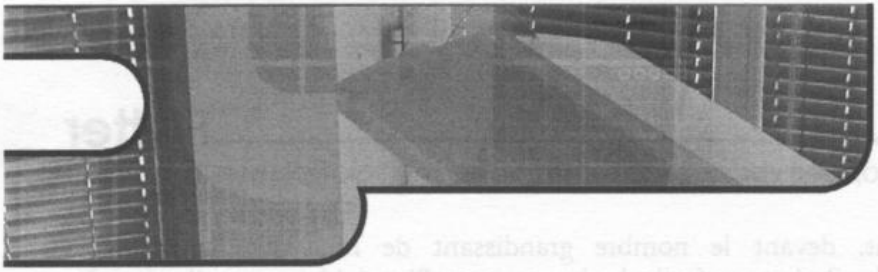
Chapitre 11
API

1. Introduction 397
2. Appareil photo 398
   2.1 Installation du package 398
   2.2 Mise à jour des permissions 398
   2.3 Le code du programme 399
   2.4 Rendu 404
   2.5 Vidéo 404
3. GPS 405
   3.1 Installation du package 405
   3.2 Mises à jour des permissions 406
   3.3 Le code du programme 406
   3.4 Rendu 409
4. Autres API 410

Chapitre 12
Données

1. Introduction 411
2. Shared Preferences 412
   2.1 Introduction 412
   2.2 Récupération des dépendances 412
   2.3 Création de l'application 412
      2.3.1 Classe Personne 413
      2.3.2 main.dart 413
   2.4 Rendu de l'application 422
3. SQLite 425
   3.1 Introduction 425
   3.2 Récupération des dépendances 426
   3.3 Création de l'application 427
      3.3.1 Classe Personne 427
      3.3.2 Classe PersonneProvider 428
      3.3.3 main.dart 435
   3.4 Rendu de l'application 444
4. JSON 446
   4.1 Définition 446
   4.2 Récupération des dépendances 448
   4.3 Création de l'application 448
      4.3.1 API 448
      4.3.2 L'application 450
   4.4 Rendu de l'application 456

Index 459



Avant-propos

1. Pourquoi cet ouvrage ?

Le monde du développement a connu de larges bouleversements ces dernières années. Certains pourraient même considérer que ce métier devient de plus en plus difficile.

En l'espace d'à peine plus de vingt ans, cette profession n'a cessé d'évoluer. Web, smartphones, objets connectés, la transition numérique s'est glissée partout.

Le problème est que, chaque nouvelle technologie apparaît, avec son lot d'outils de développement.

Ce fut le cas pour le Web avec le HTML, le CSS, le JavaScript dans un premier temps. Puis, les besoins devenant exigeants, d'autres langages ou frameworks sont venus compléter ces derniers comme TypeScript, Angular, ou React.

Ce fut également le cas pour les smartphones avec Objective-c pour iOS et Java pour Android au début, puis les évolutions ont conduit à voir apparaître respectivement Swift et Kotlin par la suite.

Et ce n'est pas tout puisque les nouvelles pratiques exigent aussi de nouvelles compétences transversales comme l'intégration continue ou les architectures micro-services.

Finalement, devant le nombre grandissant de nouveaux langages ou frameworks, il n'est pas facile de s'y retrouver. Bien évidemment, il existe des tendances. Mais il est toujours difficile de dire s'il ne s'agit pas juste d'effet de mode ou d'un choix pérenne.

1.1 Comment faire son choix ?

Dans le milieu de l'entreprise, lors de la mise en place d'un nouveau projet, le choix du *stack*, c'est-à-dire de l'ensemble des couches technologiques utilisées, est déterminant pour sa réussite.

D'abord, la question prédominante est : quelle est la ou les technologies les plus adaptées au projet, celles qui permettront de répondre aux besoins exprimés dans le cahier des charges ?

Puis, très vite, d'autres questions se posent, notamment financières. Si l'entreprise décide d'innover, elle devra vérifier si ses équipes possèdent les compétences techniques nécessaires, sinon il faudra qu'elles les acquièrent.

Si des besoins de formation sont nécessaires, sont-ils importants, longs, voire coûteux ?

Si l'entreprise doit recruter, les candidats compétents sur le marché de l'emploi sont-ils nombreux ou rares ?

Le recul sur une nouvelle technologie choisie est-il important ?

La communauté des développeurs s'est-elle emparée de cette technologie ?

La documentation est-elle suffisante, mise à jour régulièrement ?

Etc.

Toutes ces questions sont très importantes. Les entreprises, même dans le cadre de leur recherche et développement, font preuve de prudence. Une technologie peut être écartée si les garanties nécessaires ne sont pas présentes.

Toutefois, il faut aussi laisser leur chance aux nouveautés surtout quand elles sont prometteuses.

© Editions ENI - All rights reserved

1.2 Et dans l'univers du développement mobile ?

Début 2019, dans le cadre d'un projet d'application mobile, je me suis mis à chercher une solution plus aisée, plus rapide et aussi efficace que le développement de deux applications natives, une avec Java pour Android, une autre avec Swift pour iOS.

Après avoir découvert quelques outils disponibles sur le marché, mon choix s'est d'abord arrêté sur React Native. Ce n'était pas parfait, mais il n'y avait aucun doute sur le fait qu'il s'agissait du choix le plus logique.

Puis, après quelques recherches, je suis tombé sur Flutter. Et là ce fut comme une révélation. Enfin, un outil performant et agréable pour coder était proposé.

Cette découverte a suscité l'envie de partager et de faire découvrir au plus grand nombre Flutter et a conduit à la rédaction de cet ouvrage.

2. À qui s'adresse cet ouvrage ?

Ce livres s'adresse à tous ceux qui sont désireux d'apprendre une nouvelle technologie permettant la création d'applications mobiles sur n'importe quelle plateforme Android ou iOS, et cela grâce à un seul code.

Il a été conçu de telle sorte qu'il n'est pas nécessaire d'avoir des bases solides en programmation. Les notions courantes seront développées au fur et à mesure. Par exemple, le développement orienté objet sera abordé lors du chapitre Dart, notions avancées. Les débutants n'auront donc pas de difficultés à suivre.

Évidemment des notions plus avancées seront vues aussi, permettant aux développeurs plus chevronnés de découvrir en profondeur Flutter.

3. Structure de l'ouvrage

L'ouvrage présentera d'abord un état du monde du développement mobile aujourd'hui. Par la suite, la mise en place des outils sera détaillée et centrée sur les logiciels gratuits afin que chacun puisse tester et apprendre sereinement. Après ces installations, il sera temps de lancer notre première application mobile Flutter afin d'en découvrir les rouages de base.

Pour aller plus loin, il faudra prendre le temps de découvrir le langage Dart sur lequel Flutter s'appuie. Cet apprentissage se déroulera en deux temps. Les notions de base seront abordées en premier lieu, puis les notions plus complexes telles que le développement orienté objet seront vues ensuite.

Une fois rodé sur le Dart, il restera évidemment à découvrir Flutter en profondeur. Nous verrons qu'il s'appuie sur un système de *widgets* (tout est widget dans Flutter). Ceux de base seront d'abord décrits puis, avant de parler des widgets plus actifs, l'ouvrage expliquera les notions fondamentales que sont les états et la navigation.

Par la suite, afin de mieux organiser les projets, il faudra aborder les listes et les grilles.

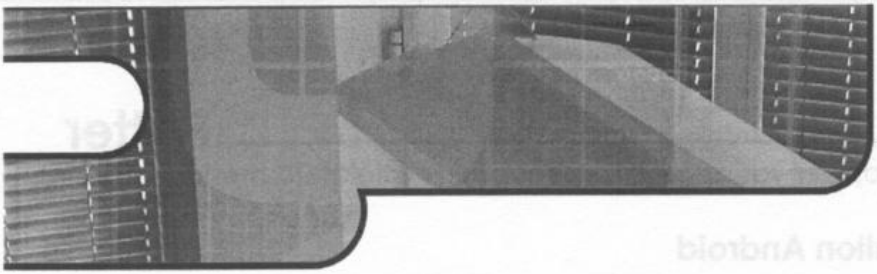
Afin de pousser l'apprentissage plus loin, l'utilisation d'API pour gérer l'appareil photo ou le GPS sera expliquée.

Le livre se terminera avec la gestion des données sous différentes formes.

Remarque

Flutter™ et le logo associé sont des marques de commerce de Google LLC. Nous ne sommes ni approuvés ni affiliés à Google LLC.

© Editions ENI - All rights reserved



Chapitre 1
Introduction

1. État des lieux du développement mobile

1.1 Un marché en progression

Le développement mobile est un marché très important. Selon le rapport d'App Annie, en 2018, il y aurait eu 194 milliards de téléchargements dans le monde tous stores d'applications confondus.

Cela représente une progression de 35 % en deux ans, ce qui reste considérable. Il y a donc un vrai potentiel au même titre que le marché du Web.

De ce fait, les offres d'emploi dans le domaine ne sont pas rares. Mais comme toujours, tout dépend des compétences possédées.

En effet, il existe, là aussi, une multitude de façons de créer des applications. La plus courante est le développement dans un langage natif soit sur Android, soit sur iOS. Il existe également beaucoup de façons de créer des applications pour les différentes plateformes à partir d'un seul et unique code.

1.2 Application Android

Concernant Android, il existe deux langages possibles : Java et Kotlin.

Le premier a longtemps été le plus prisé des entreprises puisque ce langage très courant ne demandait souvent pas de formation supplémentaire aux équipes. En effet, il y a sur le marché du travail beaucoup de développeurs Java. Quant à Kotlin, qui est beaucoup plus récent (à ce sujet, lire l'excellent ouvrage d'Anthony COSSON : Kotlin, les fondamentaux du développement d'applications Android aux Éditions ENI), il a eu plus de mal à s'imposer.

Toutefois, lors des dernières conférences Google IO qui se sont tenues courant mai 2019, Kotlin est devenu le moyen privilégié pour le développement d'applications Android. Cette position de Google ne fait qu'entériner les choses. Depuis longtemps déjà, officieusement, Java avait perdu son trône.

Plusieurs raisons peuvent expliquer cela. Tout d'abord, il existe un partenariat fort entre Google et JetBrains, le propriétaire de Kotlin. L'interface de développement (IDE) Android Studio est d'ailleurs fournie par ces derniers. De plus, ce langage est plus moderne que Java.

Mais la raison principale est qu'Oracle a entamé depuis 2010 un procès à Google pour atteinte au copyright.
Après de multiples rebondissements, Google a perdu. Reste à connaître le montant des dommages qu'ils devront payer. Mais, il est évident que les sommes seront élevées.

Autant de bonnes raisons, donc, pour laisser tomber Java au profit de Kotlin.

1.3 iOS

Apple, de son côté, a une philosophie très différente de Google. Plus refermée sur elle-même, l'entreprise a la volonté de contrôler toute sa chaine de production. Des produits, en passant par le système d'exploitation et jusqu'aux langages de programmation.

© Editions ENI - All rights reserved

Objective-c est sorti en 1983. Un peu vieillissant, il n'était pas étonnant qu'un autre lui succède. Swift est donc arrivé en 2014. Sa syntaxe est plus proche de Java ou C#.

1.4 Le multiplateforme

Pas besoin d'être un génie pour constater la difficulté imposée par cette situation bipartite.

Sur un projet qui nécessite un développement sur les deux plateformes iOS et Android, il faut donc multiplier les temps de production ainsi que les compétences.

Cette complexité a un coût très important. Logiquement, plusieurs projets ont vu le jour. Chacun a la volonté de faire plus simple et plus économique en permettant qu'avec un seul développement on puisse obtenir une application pour chaque système.

Parmi les plus connus et les plus utilisés, il y a Ionic, Xamarin et React Native.

Bien que les deux premiers soient en perte de vitesse de manière globale, il faut relever que React Native a su séduire un public large. Ses atouts résident déjà dans le langage de base qui est JavaScript. Il y a donc beaucoup de développeurs qui possèdent ces compétences. Le produit est assez performant malgré un certain manque de maturité.

Il est toujours intéressant de noter que des entreprises très importantes comme AirBnB qui avait une partie de son application codée en React Native ont fait marche arrière.

Alors que reste-t-il ?

Évidemment, Flutter.

2. Flutter

2.1 Présentation

L'arrivée de Flutter provoque un séisme dans le développement mobile. Officiellement, la version 1.0 est sortie en décembre 2018. La technologie est donc très récente.

De quoi s'agit-il ?
Flutter est un SDK (*Software Development Kit*) ou kit de développement en français. En d'autres mots, c'est un trousseau, une caisse à outils qui va faciliter le développement.
Comme cela a été détaillé dans la section Application Android de ce chapitre, les moyens de création d'applications mobiles sur Android ne sont pas la propriété de Google. Quand bien même Kotlin est poussé en avant, il n'en reste pas moins que ce langage n'appartient pas non plus à Google. Autrement dit, jusqu'à l'arrivée de Flutter, Google ne disposait d'aucun outil permettant la création d'applications qui devait se reposer sur des entreprises tierces.

Il lui fallait donc avoir ses propres outils.
Il ne faut pas oublier non plus leur projet d'un nouvel OS (système d'exploitation) maison nommé Fuchsia qui a l'ambition de remplacer à terme Android et Chrome OS. Ce nouveau système a une visée cross-platform, ce qui signifie qu'il pourrait fonctionner sur n'importe quel type d'appareils.
La firme américaine a voulu dès le début contrôler la chaine de production des applications qui pourraient être développé dessus.

Flutter est donc né de ces besoins.

Mais l'idée va plus loin. Car si Fuchsia se veut cross-platform, il fallait que Flutter le soit aussi.

© Editions ENI - All rights reserved

Au constat de ce qui a été évoqué précédemment, il restait à choisir une technologie qui permette cela. JavaScript n'a pas été retenu. La réponse se trouve peut-être dans un document interne paru en novembre 2010 nommé "Future of JavaScript" doc from our internal "JavaScript Summit" où Google fait état de son inquiétude vis-à-vis de JavaScript et de son avenir en raison, entre autres, de son manque de structure. Il indique notamment deux pistes qui pourraient être envisagées. La première, la plus facile, serait de contribuer à l'amélioration de celui-ci. Dans la mesure où cette option reste limitée, il est évoqué, dans un second temps, d'élaborer un nouveau langage qui viendrait remplacer JavaScript en comblant ses défauts.

C'est à ce titre que Google a sorti le langage Dart en 2011. Flutter s'appuie sur Dart comme langage de base, il a donc une structure solide.

Ce sujet sera abordé un peu plus loin dans la section Dart de ce chapitre.

2.2 Historique et versions

Voici les différentes versions stables de Flutter :

Version	Date de sortie
1.0	04/12/2018
1.2.1	26/02/2019
1.5.4	07/05/2019
1.7.8	24/07/2019
1.9.1	10/09/2019

Ce qui est remarquable, c'est le nombre élevé de versions sorties en si peu de temps. Cela montre l'engouement de Google vis-à-vis du projet.

D'autant plus que chaque mise à jour apporte son lot de corrections et de nouveautés.

Par exemple, lors des conférences Google IO qui se sont déroulées en mai 2019, Google a sorti la version 1.5 de Flutter dans laquelle le projet *Hummingbird* a été révélé. Il permet à Flutter de ne plus se cantonner au développement mobile mais de toucher également au Web !

Dans la dernière version sortie en septembre 2019, il y a eu plus de 1540 contributions, corrections ou améliorations.

Le support pour les derniers OS d'Apple, MacOs Catalina et iOS 13 a été rendu effectif alors que ceux-ci sont très récents ou même pas encore officiellement mis en production (Catalina date de juin 2019 et iOS 13 du 19 septembre 2019).

2.3 Dart

Logo de Dart

Dart est le langage choisi par Google pour les applications Flutter. Comme cela a été évoqué précédemment, il a d'abord été conçu pour supplanter JavaScript.

Sorti en 2011, il n'a pas finalement eu le succès escompté. Malgré tout, il est intéressant de noter qu'il est devenu une norme de l'**ECMA** (*European Computer Manufacturers Association*) au même titre que JavaScript.

Tombé un peu aux oubliettes, Dart refait surface avec l'arrivée de Flutter et se refait une nouvelle jeunesse avec l'arrivée d'une version 2.0.

C'est à ce titre qu'en septembre 2019, Google sort la version 2.5.

Dart est un langage qui a été conçu avec l'optique d'être facilement appréhendable par les développeurs maîtrisant déjà les langages C, C++ ou encore Java.

Les chapitres Dart, les fondamentaux et Dart, notions avancées permettront de détailler ses spécificités.

© Editions ENI - All rights reserved

2.4 Développement cross-platform

Le point fort incontestable de Flutter est la possibilité de pouvoir réaliser des applications pour différents supports tels qu'Android et iOS à partir d'un seul et unique code source.

Le problème avec ce type de technologie reste dans la part de code « spécifique » à adapter sur chaque OS qui peut être plus ou moins conséquente.

Avec Flutter, il n'y a quasiment aucun reste à charge. Pourquoi ? Cela est dû à la façon dont Flutter est constitué. En effet, il n'utilise pas les composants natifs mais l'ensemble de ces éléments a été recréé. Dès lors, le développeur dispose d'un ensemble de fonctionnalités (boutons, champs, dispositions...) propres à chaque plateforme (voir juste après Organisation par Widgets).

Il n'est pas faux de dire qu'une application peut être écrite à partir d'un seul et unique code.

Toutefois, il reste quelques manipulations particulières qui ne relèvent pas vraiment du code mais plus de la configuration, comme les autorisations de localisation par exemple, et évidemment les *builds* permettant d'obtenir une application finale pour chaque support.

2.5 Le hot reload

Google a pensé Flutter comme un produit qui doit séduire les développeurs, mais aussi les entreprises afin qu'il soit massivement utilisé.

Il fallait donc optimiser certains points. Le rendu est un des éléments délicats du développement mobile. En effet, les modifications du code obligent à relancer l'émulateur (ou au moins l'application) afin de voir les changements s'opérer. Dans tous les cas, comme l'application est relancée à son point de départ, il faut naviguer jusqu'à atteindre l'écran affecté par les corrections.

À chaque fois, les développeurs doivent attendre plusieurs secondes. Il est aisé de comprendre que finaliser une page avec tous ses aspects graphiques qui confinent parfois au détail est un vrai labeur.

Au bout du compte, à l'échelle d'une application entière, le temps perdu est important.

Google a répondu à cette problématique avec le *hot reload*. Toute création, changement, modification superficielle ou en profondeur est vue directement sans avoir ni à relancer l'émulateur, ni l'application.

Le gain de temps est donc énorme. C'est un avantage non négligeable face au développement natif. Les développeurs n'ont plus à s'agacer devant ces moments perdus et les entreprises voient d'un bon œil ce gain de productivité.

2.6 Organisation par widgets

Flutter présente cette particularité d'avoir une organisation par widget.

Comme il est coutume de dire, sur Flutter tout est widget !

Ainsi, il existe une multitude de widgets remplissant chacun un rôle bien précis. Pour obtenir un design de type Android, il existe un widget nommé *Material Components*. Alors que pour obtenir un style IOS, les ingénieurs de Google ont développé les widgets Cupertino.

Il existe donc des widgets pour tout et n'importe quoi. Qu'il s'agisse de texte, de champs à remplir, de barres de navigation, il y a un ou des widgets qui correspondent.

Pourquoi fonctionner avec des widgets ?

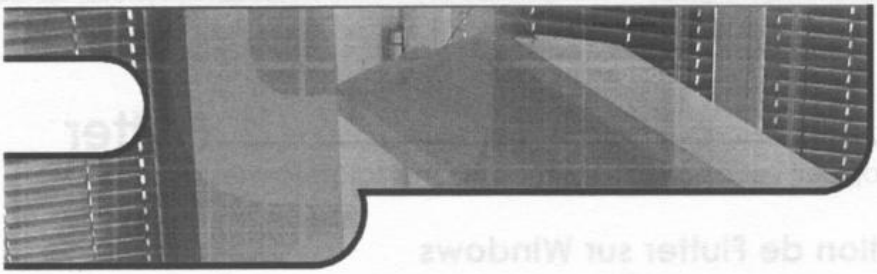
Flutter ne fonctionne pas en utilisant les composants natifs. Tout a été recodé afin d'optimiser les performances. L'avantage est que le rendu est parfaitement conforme aux éléments d'origine et qu'un seul code suffit pour réaliser une application multiplateforme.

Les widgets s'articulent un peu comme une page web et sa notion de Document Object Model (DOM). Une arborescence est créée, les widgets s'imbriquent. Il existe donc des notions de parents, d'enfants, etc.

© Editions ENI - All rights reserved

Il existe deux types de widgets, ceux qui ont un état fixe et ceux qui vont évoluer au fil du temps. Ils sont nommés respectivement *StatelessWidgets* et *StatefulWidgets*. Pour ces derniers, l'état est donc observé en permanence afin de détecter les changements et modifier en conséquence l'affichage. Ces notions seront revues en détail dans le chapitre Les états.

Afin de mettre en pratique ces aspects et de pouvoir enfin tester, il va falloir maintenant configurer notre environnement de travail et procéder à quelques installations qui vont être décrites dans le chapitre suivant.



Chapitre 2
Installation de l'environnement

1. Introduction

Il est assez aisé d'installer et de travailler avec Flutter. Les informations qui sont fournies dans ce chapitre sont valables au moment leur rédaction. Toutefois, il est bon de rappeler que le site consacré à Flutter, https://flutter.dev, est très bien conçu et est régulièrement mis à jour. Seul bémol, il est en anglais.

Dans ce chapitre, il sera question, dans un premier temps, d'installer le SDK Flutter sur les différents systèmes d'exploitation (OS) : Windows, macOS et Linux. Toutes les étapes seront décrites pas à pas. Concernant macOS, l'installation des outils utiles au développement sous iOS sera détaillée également. Une fois ces configurations effectuées, il sera temps de procéder à l'installation de l'environnement de développement (IDE) Android Studio et des plug-ins qui permettront à chacun de coder avec Flutter. Enfin, la dernière partie évoquera une ligne de commande centrale dans le SDK Flutter qui va permettre de vérifier que l'ensemble des paramètres sont au vert pour commencer à développer.

2. Installation de Flutter sur Windows

2.1 Compatibilité

La compatibilité requise pour le système d'exploitation de Microsoft est la suivante :

Flutter ne fonctionne qu'à partir de Windows 7 SP1. Il faut disposer au minimum de 400 Mo d'espaces libres sur le disque dur. Attention, ceci se rajoute aux 700 Mo nécessaires pour Android Studio, à télécharger un peu plus tard. Dans la mesure du possible, le travail sur un disque dur SSD accélérera les performances globales.

Il ne faut pas oublier de vérifier la présence de Windows PowerShell 5.0 (ou version suivante) qui est normalement préinstallé en même temps que Windows 10.

Enfin, `Git` for Windows dans sa version 2 ou supérieure est indispensable. Il est possible de le télécharger à l'adresse suivante :
https://git-scm.com/downloads

Seule recommandation, il ne faut pas oublier de cocher l'option **Use Git from the Windows Command Prompt** (Utiliser Git à partir de l'invite de commandes) à l'installation.

2.2 Téléchargement du SDK Flutter

Le SDK peut être téléchargé à partir de l'adresse suivante :
https://storage.googleapis.com/flutter_infra/releases/stable/windows/flutter_windows_v1.9.1+hotfix.4-stable.zip

Il s'agit de la version 1.9.1 valable au moment de la rédaction de l'ouvrage. Pour être certain d'avoir accès à la dernière version du SDK, il est conseillé de se rendre sur la page consacrée :
https://flutter.dev/docs/get-started/install/windows

© Editions ENI - All rights reserved

Une fois sur la page, un lien de téléchargement de ce type devrait être présent :

flutter_windows_v1.9.1+hotfix.4-stable.zip

Figure 1 - Lien de téléchargement du SDK Flutter pour Windows

2.3 Installation du SDK

Une fois la dernière version de Flutter récupérée, il ne reste plus qu'à l'installer.

Le SDK a été téléchargé sous la forme d'un fichier compressé (.zip). Il faudra donc le décompresser et choisir un dossier de destination pour les fichiers.

Remarque

Pour des raisons pratiques, il convient d'éviter les dossiers « système » ou requérant un niveau de privilèges élevés.

Par défaut, la documentation de Google propose **`C:\src\flutter`**.

À partir de ce moment, Flutter est utilisable en ligne de commandes, mais de manière limitée. Afin de ne pas subir de contraintes, il va falloir modifier le path de Windows.

Pour cela, dans la barre de recherche de Windows, il faut saisir : Modifier les variables d'environnements pour votre compte (Figure 2).

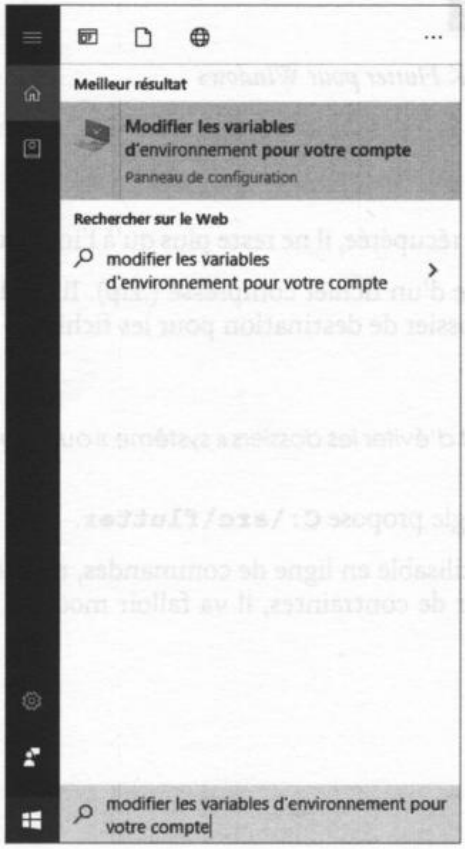


Figure 2 - Recherche Windows

© Editions ENI - All rights reserved

En cliquant sur **Modifier les variables d'environnement pour votre compte**, l'écran suivant devrait apparaître (Figure 3) :

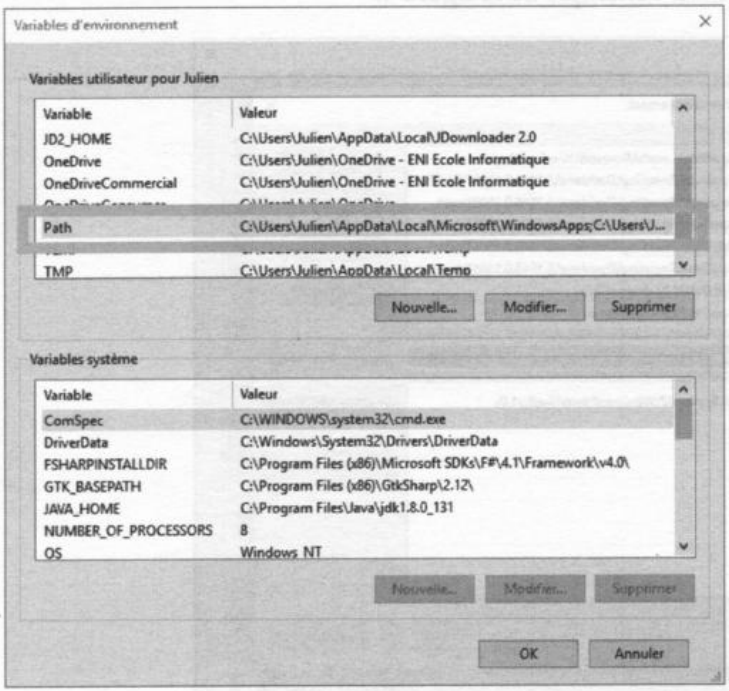


Figure 3 - Écran des variables d'environnement

Dans la zone supérieure nommée **Variables utilisateur pour XXX** (XXX étant le nom de l'ordinateur), il faudra vérifier l'existence d'une variable nommée `Path`.

Si elle existe, il suffira de la sélectionner puis de cliquer sur **Modifier**.

L'écran de modification de la variable apparaîtra. À droite de l'écran, il suffit de cliquer sur **Nouveau** puis de saisir le chemin menant au dossier « bin » de Flutter comme dans l'exemple de la figure 4.

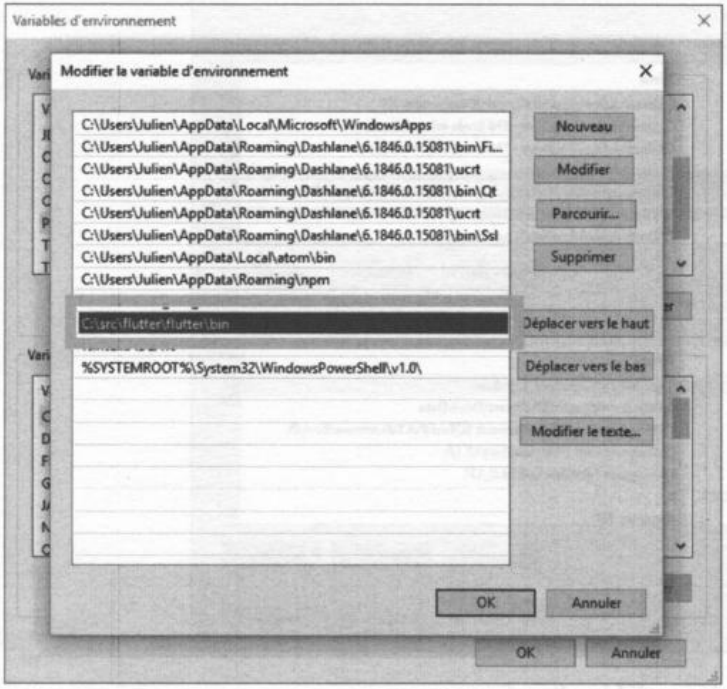


Figure 4 - Ajout du chemin vers le dossier bin de Flutter

Dans l'exemple ci-dessus, le dossier bin est situé à l'endroit même où l'archive .zip de Flutter a été décompressée, c'est-à-dire C:\src\flutter\flutter.

Si la variable d'environnement `Path` n'est pas présente, il faut la créer. Pour cela, il suffit de cliquer sur **Nouvelle...**.

© Editions ENI - All rights reserved

Une nouvelle fenêtre va apparaître. Dans le premier champ nommé **Nom de la variable**, il faudra saisir « Path » et dans le second, **Valeur de la variable**, le chemin du dossier de l'archive décompressée de Flutter (Figure 5).

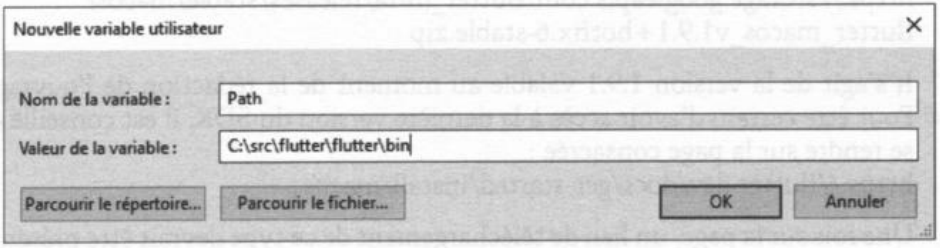


Figure 5 - Création d'une nouvelle variable d'environnement

Une fois cette étape terminée, il sera possible de lancer un terminal pour exécuter les premières commandes de Flutter. Cette partie sera abordée un peu plus loin dans la section Flutter Doctor de ce chapitre.

3. Installation de Flutter sur macOS

3.1 Compatibilité

Sur macOS version 64 bits, il faut disposer d'au moins 2,8 Go d'espace libre auxquels il faudra rajouter l'espace pour Android Studio (environ 700 Mo).

Pour l'installation, le terminal et `bash` (ou `zsh`, comme il sera expliqué un peu après) seront utiles ainsi que `vi/vim`. Heureusement, ils sont déjà installés par défaut sur l'environnement macOS.

En revanche, il faut également le gestionnaire de dépendances `cocoapods...` qui, lui, n'est pas installé. Son installation est décrite ci-après dans la section Installation de cocoapods.

3.2 Téléchargement du SDK Flutter

Le SDK peut être téléchargé à partir de l'adresse suivante :
https://storage.googleapis.com/flutter_infra/releases/stable/macos/flutter_macos_v1.9.1+hotfix.6-stable.zip

Il s'agit de la version 1.9.1 valable au moment de la rédaction de l'ouvrage. Pour être certain d'avoir accès à la dernière version du SDK, il est conseillé de se rendre sur la page consacrée :
https://flutter.dev/docs/get-started/install/macos

Une fois sur la page, un lien de téléchargement de ce type devrait être présent :

Figure 6 - Lien de téléchargement du SDK Flutter pour macOS

3.3 Installation du SDK

Après avoir téléchargé le fichier compressé du SDK Flutter (.zip), il faudra ouvrir le terminal. Pour cela, il suffit soit de cliquer sur la loupe en haut à gauche de votre écran pour activer la recherche Spotlight :

Figure 7 - Outil de recherche macOS Spotlight

© Editions ENI - All rights reserved

Puis de saisir **Terminal** dans la zone de recherche. L'autre solution est d'ouvrir le Launchpad :

Figure 8 - Launchpad MacOS

Puis de chercher le terminal qui se situe normalement dans le dossier « **Autre** » :

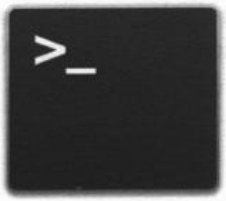


Figure 9 - Terminal MacOS

À cette étape, il va falloir choisir le dossier dans lequel le SDK va être installé. Par défaut, le dossier recommandé est « `development` ». Dans tous les cas, il faudra être capable de se souvenir de l'endroit où est installé le SDK.

Une fois le choix de l'emplacement fait, il faut s'y déplacer grâce à la commande :

```
$ cd ~/development
```

Remarque

Le symbole ~ s'obtient en combinant [option] + N.

Remarque

Le symbole $ n'est qu'une indication de début de ligne et n'est pas à recopier. Ceci est valable pour l'ensemble des instructions suivantes.

Toutefois, si ce dossier est inexistant, il faut au préalable le créer en suivant cette ligne de commande :

```
$ mkdir development
```

Attention de bien vérifier l'endroit où vous vous trouvez avant de créer le dossier.

Une fois le dossier créé et après s'y être déplacé, il faut exécuter la ligne de commande suivante qui permet de dézipper le SDK :

```
$ unzip ~/Downloads/flutter_macos_v1.9.1+hotfix.6-stable.zip
```

Flutter est installé, il reste une dernière étape afin de pouvoir s'en servir convenablement. Il va falloir l'ajouter au `path` afin de bénéficier des instructions disponibles en ligne de commande dans le terminal.

Pour cela, il suffit de saisir :

```
$ export PATH="$PATH:`pwd`/flutter/bin"
```

Remarque

Attention à bien respecter la syntaxe. Ici, les apostrophes situées autour de `pwd` sont à l'envers ([Alt Gr] 7 sur PC).

Remarque

Besoin de récupérer les ressources binaires en local ?

Dans le cadre du développement de son Framework, Google offre également la possibilité de télécharger les fichiers binaires correspondant aux différentes plateformes. Cette option s'avère utile dans un contexte de production restreint et hermétique ou si le réseau est indisponible. Dans ce cas, pour les récupérer, il faudra saisir dans le terminal la ligne de commande suivante :

```
$ flutter precache
```

Il est également possible d'obtenir des options supplémentaires pour ce téléchargement en appliquant la ligne de commande :

```
$ flutter help precache
```

© Editions ENI - All rights reserved

À ce moment, Flutter est prêt à être utilisé. D'ailleurs, les lignes de commande précédentes en sont la preuve si jamais elles ont été utilisées.

Toutefois, l'ajout dans le path effectué précédemment est temporaire. Pour l'inscrire de manière pérenne, il faut quelques opérations supplémentaires.

En fonction de la version de l'OS, ces étapes peuvent varier un peu. En réalité, tout dépend du Shell utilisé. Le Shell correspond à l'interface de ligne de commande pour laquelle l'ordinateur est configuré. Par défaut, les Mac fonctionnaient avec `bash` jusqu'à la version de macOS Mojave. À partir de la dernière mise à jour, Catalina, le Shell a changé. C'est désormais `zsh` qui sera la référence.

Toutefois, même si la mise à jour de l'OS a été effectuée vers Catalina, cela ne veut pas dire que le Shell a subi le même traitement. Dans ce cas, une petite manipulation est nécessaire pour basculer de `bash` vers `zsh`. Elle sera détaillée un peu plus loin dans la section Catalina.

Voici les manières de procéder en fonction de la version du système d'exploitation :

3.3.1 Mojave (et les versions précédentes)

Sur ces versions, le Shell par défaut est bash. Pour insérer Flutter dans le `path`, il faut créer un fichier nommé `rc` au niveau du dossier `$HOME`. `$HOME` correspond à la session en cours ou, plus simplement, à l'emplacement Users/NomUtilisateur.

Pour créer et éditer le fichier, il va falloir utiliser un éditeur de textes comme `vi` ou `vim` qui sont normalement déjà intégrés au sein de macOS. La ligne de commande suivante servira à lancer et, le cas échéant, à créer le fichier :

- Pour vi :

```
$ vi .bashrc
```

- Pour vim :

```
$ vim .bashrc
```

Grâce à cette ligne de commande, la fenêtre suivante devrait s'ouvrir :

Figure 10 - Fichier .bashrc ouvert avec l'éditeur de texte vim

Il faudra y saisir le chemin d'accès à Flutter. Pour commencer la saisie, il suffit d'appuyer sur la touche **i** du clavier.

Remarque

*Lorsque la saisie est activée grâce à la touche **i**, la mention « `insert` » apparaît en bas à gauche de l'éditeur.*

À cet instant, il faudra saisir cette ligne :

```
$ export PATH="$PATH:[Chemin_vers_le_dossier_Flutter]/flutter/bin"
```

Bien évidemment, il faut remplacer `[Chemin_vers_le_dossier_Flutter]` par le véritable chemin. Afin de procéder le plus simplement du monde, il suffit de copier le début de l'instruction. Puis, au moment d'insérer le chemin, il suffit de se placer dans le dossier de Flutter et de faire un glisser/déposer du dossier « bin » dans la fenêtre de l'éditeur. Le chemin se copie alors tout seul. Enfin, il ne faut pas oublier les guillemets qui viennent clôturer notre ligne.

© Editions ENI - All rights reserved

Quand la saisie est terminée, il faut appuyer sur [Echap] (**esc**) pour sortir du mode de saisie puis il suffit de taper ceci pour enregistrer le fichier et sortir de l'éditeur :

```
:wq !
```

Pour que l'action qui vient d'être réalisée soit prise en compte, il faut fermer le terminal puis le rouvrir. Afin de vérifier maintenant si Flutter a bien été ajouté au path, il faut saisir dans le terminal :

```
$ echo $PATH
```

Un résultat semblable à celui-ci devrait apparaître :

```
MacBook-Pro-de-Julien:~ julientrillard$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:Users/julientrillard/development/fl
utter/bin
```

Figure 11 - Résultat de la commande echo $PATH

Enfin, une dernière étape va permettre de vérifier que la commande Flutter est bien accessible :

```
$ which flutter
```

Le résultat de cette action devrait être quasi-similaire à celui de la figure 11.

Une fois cette étape terminée, il sera possible de lancer un terminal pour exécuter les premières commandes de Flutter. Cette partie sera abordée un peu plus loin dans la section Flutter Doctor de ce chapitre.

3.3.2 Catalina

Sur Catalina, il faut distinguer deux possibilités. Soit cette version de l'OS est déjà présente sur le Mac lors de l'achat, soit il s'agit d'une mise à jour.

La différence qui nous intéresse réside dans le fait que sur une version préinstallée de Catalina le Shell par défaut est `zsh` tandis que s'il s'agit d'une mise à jour, `bash` est resté paramétré comme Shell et il faudra une manipulation pour en changer. Cela tombe bien, puisque Apple recommande désormais l'utilisation de `zsh`, il convient de basculer vers celui-ci.

```
julientrillard — -bash — 80×24
Last login: Thu Nov 14 18:15:22 on ttys001

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
```

Figure 12 - Message passant dans un terminal bash pour avertir du changement de Shell vers zsh

- Pour cela, il faut cliquer sur le symbole si connu d'Apple, la pomme, situé en haut à gauche de l'écran :

Figure 13 - Barre d'outils

- Ensuite, il faudra cliquer sur **Préférences Système**. Ici, il faudra choisir **Utilisateurs et groupe** :

Figure 14 - Menu Préférences Système

© Editions ENI - All rights reserved

- Dans la nouvelle fenêtre, il faut cliquer sur le **cadenas** afin d'autoriser les modifications.

Figure 15 - Menu Utilisateurs et groupes

- À cet instant, il faut renseigner le mot de passe puis valider.
- Ensuite, de retour sur l'écran des Utilisateurs et groupes, il suffit d'effectuer [Ctrl] + clic sur l'utilisateur actuel afin d'obtenir l'accès aux options avancées.

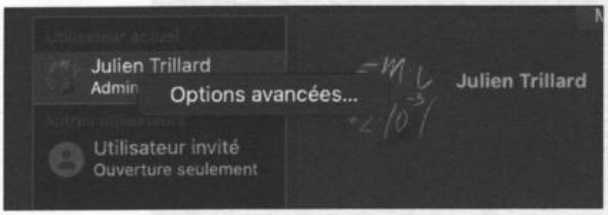


Figure 16 - Accès aux options avancées

Sur l'écran des options avancées, il convient de choisir dans la catégorie **Shell d'accès**, le Shell **/bin/zsh** puis de valider en cliquant sur **OK**.

Le Shell venant d'être configuré, il ne reste plus qu'à ajouter Flutter au `path`.

Pour insérer Flutter dans le `path`, il faut créer un fichier nommé `.zshrc` au niveau du dossier `$HOME`. `$HOME` correspond à la session en cours ou, plus simplement, à l'emplacement `Users/NomUtilisateur`.

Pour créer et éditer le fichier, il va falloir utiliser un éditeur de textes comme `vi` ou `vim` qui sont normalement déjà intégrés au sein de macOS. La ligne de commande suivante servira à ouvrir, et le cas échéant, à créer le fichier :

Remarque

Le symbole % n'est qu'une indication de début de ligne et n'est pas à recopier. Ceci est valable pour l'ensemble des instructions suivantes.

- Pour vi :

```
% vi .zshrc
```

- Pour vim :

```
% vim .zshrc
```

Grâce à cette ligne de commande, la fenêtre suivante devrait s'ouvrir :

Figure 17 - Fichier .zshrc ouvert avec l'éditeur de texte vim

© Editions ENI - All rights reserved

- Il faut y saisir le chemin d'accès à Flutter.
- Pour commencer la saisie, il suffit d'appuyer sur la touche **i** du clavier.

Remarque

*Lorsque la saisie est activée grâce à la touche **i**, la mention `insert` apparaît en bas à gauche de l'éditeur.*

- À cet instant, il faut saisir cette ligne :

```
% export PATH="$PATH:[Chemin_vers_le_dossier_Flutter]/flutter/bin"
```

- Bien évidemment, il faut remplacer `[Chemin_vers_le_dossier_Flutter]` par le véritable chemin.
- Afin de procéder plus simplement, il suffit de copier le début de l'instruction. Puis, au moment d'insérer le chemin, il suffit de se placer dans le dossier de Flutter et de faire un glisser/déposer du dossier « bin » dans la fenêtre de l'éditeur. Le chemin se copie alors tout seul.
- Enfin, il ne faut pas oublier les guillemets qui viennent clôturer notre ligne.
- Quand la saisie est terminée, il faut appuyer sur [Echap] (**esc**) pour sortir du mode de saisie puis il suffit de taper ceci pour enregistrer et sortir de l'éditeur :

```
:wq !
```

- Pour que l'action qui vient d'être faite soit prise en compte, il faut fermer le terminal puis le rouvrir. Afin de vérifier maintenant si Flutter a bien été ajouté au `path`, il faut saisir dans le terminal :

```
% echo $PATH
```

Un résultat semblable à celui-ci devrait apparaître :

```
julientrillard@MacBook-Pro-de-Julien ~ % echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Users/julientrillard/development/f
lutter/bin
```

Figure 18 - Résultat de la commande echo $PATH

Enfin, une dernière étape va permettre de vérifier que la commande Flutter est bien accessible :

```
% which flutter
```

Le résultat de cette action devrait être quasi-similaire à celui de la figure 18.

Une fois cette étape terminée, il sera possible de lancer un terminal pour exécuter les premières commandes de Flutter. Cette partie sera abordée un peu plus loin dans la section Flutter Doctor de ce chapitre.

3.4 Installation de cocoapods

Cocoapods est un gestionnaire de dépendances qui va être utile. De manière générale, un gestionnaire de dépendances sert à installer, comme son nom l'indique, des dépendances, comme des bibliothèques par exemple, simplement en les spécifiant au lieu d'avoir à installer à la main les fichiers sources de ces dernières.

Sur Mac, la ligne de commande qui permet d'obtenir cocoapods est :

```
% sudo gem install cocoapods
```

En principe, l'installation est assez rapide.

3.5 Installation de Xcode

Afin de pouvoir produire des applications pour iOS, il convient d'installer Xcode. Cet outil sera utile pour la création de l'application, mais aussi pour avoir accès à un émulateur iPhone ou pour lancer l'application directement sur un produit équipé d'iOS.

© Editions ENI - All rights reserved

Pour cela, rien de plus simple. Il suffit de se rendre dans l'App Store :

Figure 19 - Logo de l'App Store

- Dans la zone de recherche, il suffit de saisir « Xcode » et de suivre les différentes étapes d'installation.

Figure 20 - Logo Xcode

Afin d'être certain d'utiliser la dernière version installée de Xcode, il peut être utile de saisir les lignes de commande suivantes :

```
% sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer
% sudo xcodebuild -runFirstLaunch
```

Normalement, il s'agit du chemin qui fonctionnera dans la plupart des cas. Néanmoins, si l'utilisation d'une version plus ancienne est nécessaire, il faut modifier le chemin de la ligne de commande ci-dessus.

Remarque

Il est recommandé d'utiliser la dernière version stable de Xcode. Dans la pratique, les anciennes versions peuvent fonctionner, mais l'optimisation ne sera pas parfaite.

Pour finaliser l'installation de Xcode, il est nécessaire de l'ouvrir au moins une fois afin de valider le contrat de licence proposé. Il existe également une ligne de commande qui permet de réaliser cette action :

```
% sudo xcodebuild -license
```

4. Installation de Flutter sur Linux

4.1 Compatibilité

Afin de pouvoir travailler avec Flutter sur Linux, il faut disposer d'un système 64 bits. L'espace de stockage nécessaire est d'environ 600 Mo pour le SDK. Il convient, en outre, de compter l'espace pour l'IDE Android Studio soit un peu plus de 730 Mo pour la dernière version en cours.

Question outils, il faut un Shell (`bash`, `zsh`) permettant l'exécution des lignes de commande courantes. Git devra également être installé. Si ce n'est pas le cas, il convient d'exécuter les lignes de commande suivante :

```
$ sudo apt-get update
$ sudo apt-get install git-core
```

La première ligne met à jour le système, la seconde télécharge et installe Git.

Afin de vérifier que l'installation a bien été réalisée, il est possible de taper cette ligne :

```
$ git -version
```

Le résultat suivant devrait s'afficher (le numéro de version ne sera pas forcément le même) :

```
trillard@trillard-Ubuntu:~$ git --version
git version 2.17.1
```

Figure 21 - Vérification de la version de git

© Éditions ENI - All rights reserved

Dernière information importante, la commande « test » de Flutter aura besoin de la bibliothèque libGLU.so.1. La ligne de commande permettant la récupération de cette dernière est :

```
$ sudo apt-get install libglu1
```

Remarque

Le symbole $ n'est qu'une indication de début de ligne et n'est pas à recopier. Ceci est valable pour l'ensemble des instructions suivantes.

4.2 Téléchargement du SDK Flutter

Le SDK peut être téléchargé à partir de l'adresse suivante :
https://storage.googleapis.com/flutter_infra/releases/stable/linux/flutter_linux_v1.9.1+hotfix.6-stable.tar.xz

Il s'agit de la version 1.9.1 valable au moment de la rédaction de l'ouvrage. Pour être certain d'avoir accès à la dernière version du SDK, il est conseillé de se rendre sur la page consacrée :
https://flutter.dev/docs/get-started/install/linux

Une fois sur la page, un lien de téléchargement de ce type devrait être présent :

flutter_linux_v1.9.1+hotfix.6-stable.tar.xz

Figure 22 - Lien de téléchargement du SDK Flutter pour Linux

4.3 Installation du SDK

Après avoir téléchargé le fichier compressé du SDK Flutter (.tar.xz), il faut ouvrir le terminal en tapant le raccourci-clavier « [Ctrl][Alt] + T ».

À cette étape, il s'agit de choisir le dossier dans lequel le SDK va être installé. Par défaut, le dossier recommandé est « `development` ». Dans tous les cas, il faudra être capable de se souvenir de l'endroit où est placé le SDK.

Une fois le choix de l'emplacement fait, il faudra se déplacer dedans grâce à la commande :

```
$ cd ~/development
```

Remarque

Le symbole ~ s'obtient en faisant [AltGr] + 2.

Remarque

Le symbole $ n'est qu'une indication de début de ligne et n'est pas à recopier. Ceci est valable pour l'ensemble des instructions suivantes.

Toutefois, si ce dossier est inexistant, il faudra au préalable le créer en suivant cette ligne de commande :

```
$ mkdir development
```

Attention avant de créer le dossier de bien vérifier l'endroit où vous vous trouvez.

Une fois le dossier créé et après s'y être déplacé, il faudra exécuter la ligne de commande suivante qui permettra de dézipper le SDK :

```
$ tar xf ~/Téléchargements/flutter_linux_v1.9.1+hotfix.6-stable.tar.xz
```

Flutter est installé, il reste une dernière étape à franchir afin de pouvoir s'en servir convenablement. Il va falloir l'ajouter au `path` afin de bénéficier des instructions disponibles en ligne de commande dans le terminal.

Pour cela, il suffit de saisir :

```
$ export PATH="$PATH:`pwd`/flutter/bin"
```

Attention à bien respecter la syntaxe. Ici, les apostrophes situées autour de pwd sont à l'envers ([Alt Gr] 7 sur PC).

© Editions ENI - All rights reserved

Remarque

Besoin de récupérer les ressources binaires en local ?

Google offre également, dans le cadre du développement de son Framework, la possibilité de télécharger les fichiers binaires correspondants aux différentes plateformes. Cette option s'avère utile dans un contexte de production restreint et hermétique ou si le réseau est indisponible. Dans ce cas, pour les récupérer, il faudra saisir dans le terminal la ligne de commande suivante :

```
$ flutter precache
```

Il est également possible d'obtenir des options supplémentaires dans le cadre de ce téléchargement en appliquant la ligne de commande :

```
$ flutter help precache
```

À ce moment, Flutter est prêt à être utilisé. D'ailleurs les lignes de commande précédentes en sont la preuve si jamais elles ont été utilisées.

Toutefois, l'ajout dans le `path` effectué précédemment est temporaire. Pour l'inscrire de manière pérenne, il faut procéder à quelques actions supplémentaires.

Normalement, il existe un fichier nommé `rc` dans lequel on inscrit les ajouts au `path`.

Dans tous les cas, afin de modifier le path de manière pérenne et avoir ainsi accès à la commande Flutter dans le terminal, la ligne de commande suivante devra être utilisée :

```
$ export PATH="$PATH:[Chemin_vers_le_dossier_Flutter]/flutter/bin"
```

Bien évidemment, il faut remplacer `[Chemin_vers_le_dossier_Flutter]` par le véritable chemin. Cela peut donner par exemple ceci :

```
trillard@trillard-Ubuntu:~$ export PATH="$PATH:/home/trillard/development/flutt
er/bin"
```

Figure 23 - Exemple de ligne de commande pour ajouter Flutter au path définitivement

Pour s'assurer que l'opération a correctement fonctionné, il est possible de saisir cette commande :

```
$ echo $PATH
```

Un affichage de ce qui est contenu dans le `path` va s'afficher et le chemin relatif à Flutter devrait s'y trouver.

Il reste maintenant à vérifier que la commande Flutter s'exécute bien. Pour cela, il est possible de vérifier la version du SDK Flutter :

```
$ flutter -version
```

```
trillard@trillard-Ubuntu:~$ flutter --version
Flutter 1.9.1+hotfix.6 • channel stable •
https://github.com/flutter/flutter.git
Framework • revision 68587a0916 (il y a 2 mois) • 2019-09-13 19:46:58 -0700
Engine • revision b863200c37
Tools • Dart 2.5.0
```

Figure 24 - Vérification de la version de Flutter

Une fois cette étape terminée, il sera possible de lancer un terminal pour exécuter les premières commandes de Flutter. Cette partie sera abordée un peu plus loin dans la section Flutter Doctor de ce chapitre.

5. Mise à jour du SDK

Une fois que les installations sont réalisées, peu importe la plateforme (Windows, macOS ou Linux), il faudra de temps en temps mettre à jour la version installée du SDK.

Cela suppose donc deux choses. La première est de connaître la version installée et la seconde d'être averti de la disponibilité d'une mise à jour.

5.1 Connaître la version de Flutter installée

Pour retrouver la version utilisée, il faut ouvrir un terminal et indiquer la ligne de commande :

```
$ flutter --version
```

© Editions ENI - All rights reserved

Le résultat est de cette action est :

```
C:\Users\Julien>flutter --version
Flutter 1.9.1+hotfix.6 • channel stable • https://github.com/flutter/flutter.git
Framework • revision 68587a0916 (3 months ago) • 2019-09-13 19:46:58 -0700
Engine • revision b863200c37
Tools • Dart 2.5.0
```

Figure 25 - Résultat de `flutter -- version`

Il est possible de connaître la version de Flutter qui apparaît sur la première ligne de résultat ainsi que la version de Dart, sur la dernière.

5.2 Être au courant des nouveautés et des mises à jour

Pour être au courant des dernières nouveautés du SDK et des sorties d'éventuelles mises à jour, il y a plusieurs possibilités.

En premier lieu, le site de Flutter est certainement le meilleur endroit : https://flutter.dev/

En dehors, les équipes de Google communiquent beaucoup par l'intermédiaire des réseaux sociaux :

- Twitter : https://twitter.com/flutterdev
- YouTube : https://www.youtube.com/flutterdev

Il reste enfin la page GitHub du projet : https://github.com/flutter

Dans tous les cas, l'exécution de la commande `flutter --version` avertit également de la possibilité d'une mise à jour grâce à un message (cf. Figure 26).

```
C:\Users\TRILLARDJULIEN>flutter --version

  A new version of Flutter is available!

  To update to the latest version, run "flutter upgrade".

Flutter 1.9.1+hotfix.2 • channel stable • https://github.com/flutter/flutter.git
Framework • revision 2d2a1ffec9 (3 months ago) • 2019-09-06 18:39:49 -0700
Engine • revision b863200c37
Tools • Dart 2.5.0
```

Figure 26 - Message d'avertissement de la disponibilité d'une mise à jour du SDK

5.3 Faire la mise à jour du SDK

Pour procéder à la mise à jour, il suffit d'ouvrir le terminal et de taper la commande :

```
$ flutter upgrade
```

À l'issue, `flutter doctor` sera automatiquement exécutée. Cette étape est détaillée dans la section Flutter Doctor de ce chapitre.

6. Installation d'Android Studio

6.1 Introduction

Pour pouvoir produire du code, il va falloir un IDE (interface de développement) qui prenne en charge Flutter.

À ce jour, plusieurs possibilités existent : Visual Studio Code (VS Code), IntelliJ ou Android Studio.

Dans cet ouvrage, nous travaillerons avec ce dernier. Afin de suivre, il sera plus simple pour les débutants de travailler avec. Pour ceux qui possèdent déjà des notions, il n'y a aucun problème à utiliser un des deux autres IDE.

6.2 Pourquoi choisir Android Studio?

Android Studio est édité par JetBrains. Il repose sur des mécanismes et une approche identique à ceux d'IntelliJ. Dans la mesure où les applications Android ont d'abord disposé de cette plateforme de développement, Google a conservé la tradition avec Flutter. En effet, l'objectif premier est bien de développer des applications mobiles.

L'avantage d'Android Studio réside principalement dans la facilité d'installation des outils de Flutter. Mais aussi dans l'aide précieuse que pourra fournir cet IDE pendant les phases de développement. Cet ouvrage s'adressant à un large public, il convient d'aider aussi ceux qui en auraient le plus besoin.

© Editions ENI - All rights reserved

Dernier élément et non des moindres, il est totalement gratuit. En résumé, il est donc possible de créer des applications mobiles avec un investissement minimal puisqu'il n'y a rien à payer ni pour l'IDE, ni pour le SDK qui est open source.

6.3 Installation d'Android Studio

La version à télécharger est la 3.0 ou suivante.

- Pour cela, il suffit de se rendre à l'adresse suivante : https://developer.android.com/studio
- Ou de saisir dans un moteur de recherche « **`Android Studio download`** ».
- Et de cliquer sur le lien :

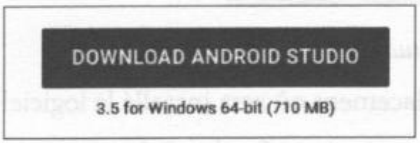


Figure 27 - Lien de téléchargement d'Android Studio

- Acceptez ensuite les conditions d'utilisation si elles vous conviennent, puis cliquez sur **DOWNLOAD ANDROID STUDIO**.
- Il ne restera plus qu'à enregistrer le fichier. Le téléchargement peut s'avérer assez long. La version 3.5 par exemple pèse plus de 700 Mo.

Une fois le téléchargement terminé, il faudra exécuter le programme d'installation.

Sur l'écran de la figure 28, il faudra absolument laisser coché **Android Virtual Device**. Il s'agit là de l'outil d'émulation d'un appareil Android sur votre ordinateur. Il est vraiment nécessaire afin de tester les applications développées.

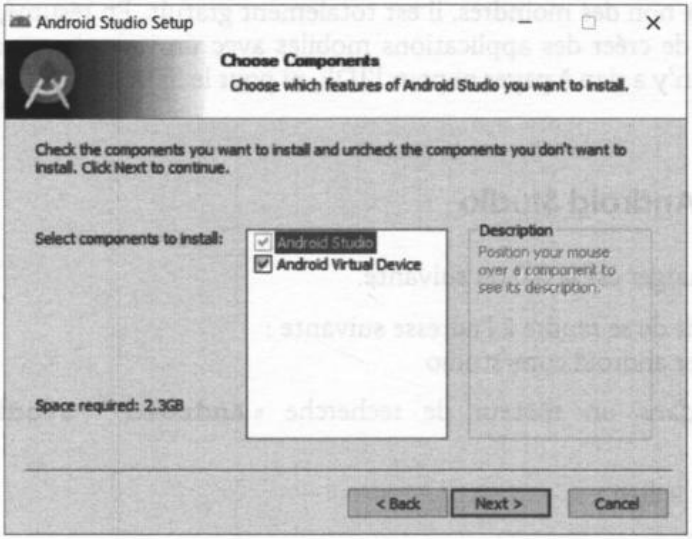


Figure 28 - Écran d'installation d'Android Studio

Après cette étape, il faudra choisir l'emplacement où sera installé le logiciel.

Sur le dernier écran, il est proposé de lancer Android Studio. Il faut accepter.

À ce stade, l'installation se poursuit.

- Au premier écran, il suffira de cliquer sur **Next** ou **Suivant**. Au deuxième écran, il sera demandé quel type d'installation vous voulez effectuer. Laisser le choix sur **Standard** et cliquez sur **Suivant**.

L'étape d'après est purement esthétique. Il faudra choisir un thème pour votre interface. Le choix propose un mode clair ou un mode sombre qui sont nommés respectivement IntelliJ ou Darcula.

Il ne reste plus qu'à cliquer sur **Finish** ou **Terminer** au dernier écran.

En fonction de la puissance de votre machine, l'installation ne devrait pas être trop longue.

Une fois celle-ci terminée, la page d'accueil d'Android Studio va s'afficher. Il ne reste plus qu'à installer les plug-ins de Flutter et Dart.

© Editions ENI - All rights reserved

6.4 Installation des plug-ins Flutter et Dart

De base, Android Studio n'est pas prévu pour fonctionner avec Flutter. Il va donc falloir ajouter un plug-in.

- Sur l'écran d'accueil (figure 29), il faudra cliquer sur **Configure** en bas à droite.

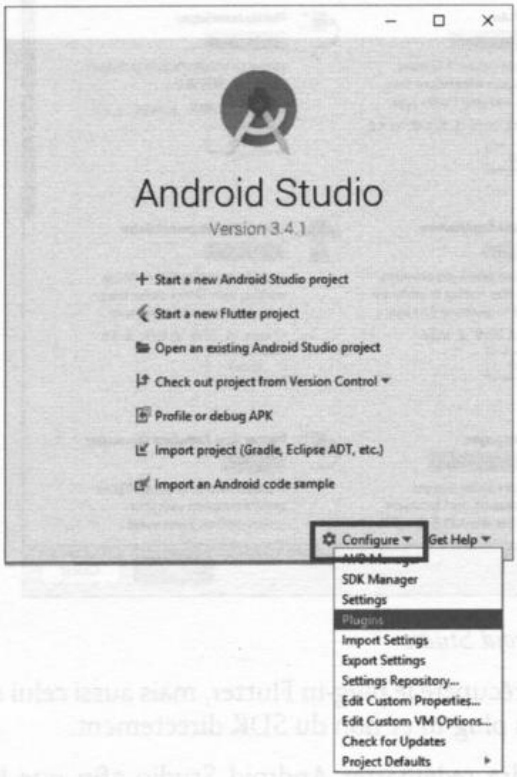


Figure 29 - Ecran d'accueil d'Android Studio 3.5

- Après, il faudra sélectionner **Plugins**.

- Puis saisir « flutter » dans la zone de recherche. Les propositions suivantes devraient apparaître (figure 30) :

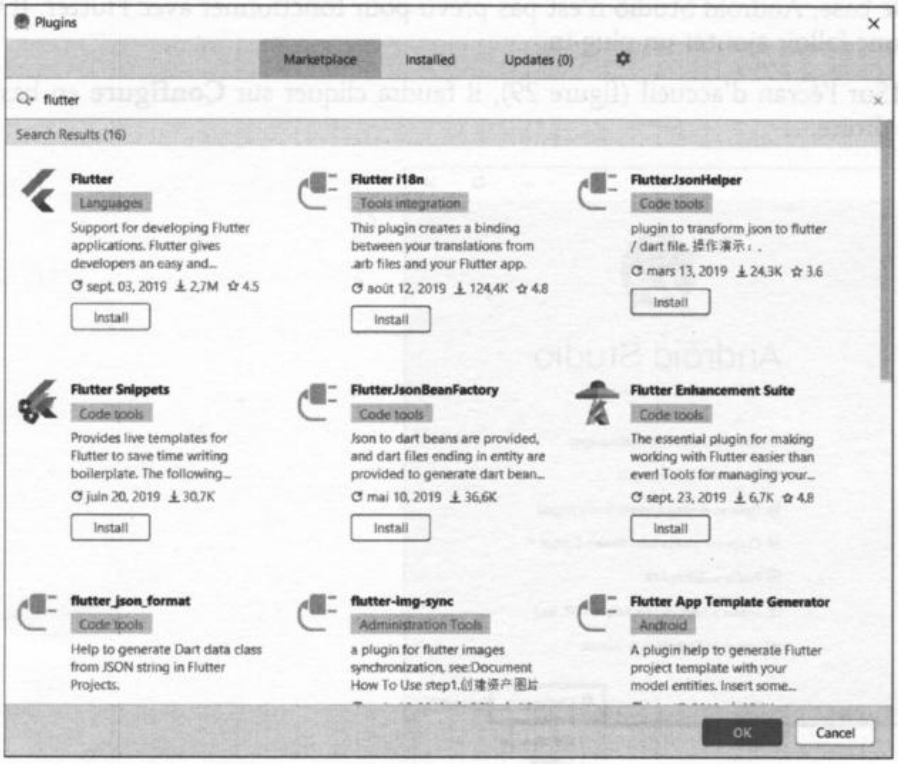


Figure 30 - Plugin Flutter sous Android Studio

À cette étape, le téléchargement récupère le plug-in Flutter, mais aussi celui de Dart. Attention, il s'agit juste du plug-in et non du SDK directement.

Après le téléchargement, il faudra redémarrer Android Studio afin que les plug-ins soient pris en compte.

Une fois Android Studio redémarré, il ne reste plus qu'une étape de configuration pour utiliser le SDK, Flutter Doctor.

© Editions ENI - All rights reserved

7. Flutter Doctor

7.1 Introduction

Flutter Doctor est une ligne de commande créée pour faciliter la vie des développeurs pendant l'installation et la configuration du SDK Flutter.

Cet outil va permettre une analyse des éléments indispensables au bon fonctionnement du produit.

Avant de commencer, si ce n'est pas le cas, il faut ouvrir Android Studio.

▶ Pour l'utiliser, il faut ouvrir un terminal du système d'exploitation et saisir :

```
$ flutter doctor
```

Un résultat semblable à celui-ci devrait s'afficher :

```
Running flutter doctor...
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.9.1+hotfix.6, on Microsoft Windows [version 10.0.17763.864], locale fr-FR)
[!] Android toolchain - develop for Android devices (Android SDK version 28.0.3)
    ! Some Android licenses not accepted.  To resolve this, run: flutter doctor --android-licenses
[!] Android Studio (version 3.5)
    X Flutter plugin not installed; this adds Flutter specific functionality.
    X Dart plugin not installed; this adds Dart specific functionality.
[!] Connected device
    ! No devices available

! Doctor found issues in 3 categories.
```

Figure 31 - Résultat de l'exécution de flutter doctor

7.2 Analyse de Flutter Doctor

Sur la figure 31, l'analyse de la commande `flutter doctor` révèle plusieurs choses.

D'abord, elle vérifie la version de Flutter actuellement installée. Ici, cette étape est validée par un tag (✓). Cela confirme que l'installation réalisée au cours des étapes 2, 3 ou 4 s'est bien passée.

Ensuite, Flutter Doctor vérifie que la licence Android permettant de développer sur des produits liés à cet OS est bien acceptée.

Troisième point, Flutter Doctor a détecté la présence d'Android Studio et a vérifié que les plug-ins de Flutter (Flutter et Dart) sont bien installés. Cette étape devrait normalement être cochée si les consignes d'installation d'Android Studio prévues dans la section Installation d'Android Studio du présent chapitre ont bien été suivies.

Dernière vérification, la présence d'un téléphone mobile, d'une tablette ou d'un émulateur doit être détectée.

Il est également possible que d'autres vérifications que celles décrites ici soient effectuées. Notamment si un autre IDE est utilisé ou encore en fonction de l'OS comme sur Mac, par exemple, où la présence de cocoapods sera vérifiée.

7.3 Corrections et validations

Il convient maintenant de corriger les éléments posant problème. Théoriquement, il devrait n'en rester que deux si toutes les étapes décrites précédemment ont été respectées. Il s'agira normalement de la licence Android et de la mise en place d'un appareil physique ou d'un émulateur pour le rendu des applications.

7.3.1 Licence Android

Afin d'accepter la licence Android, il suffit de lancer la commande :

```
$ flutter doctor -android-licenses
```

Une validation sera demandée afin d'accepter les conditions prévues par Google concernant la licence Android.

© Editions ENI - All rights reserved

7.3.2 Plug-ins et autres IDE (Visual Studio Code ou IntelliJ)

Si la seconde vérification concernant la présence des plug-ins Flutter et Dart dans Android Studio n'est pas correcte, il suffira de reprendre les conseils prodigués à la fin de la section Installation d'Android Studio de ce chapitre.

Remarque

Si d'autres IDE comme Visual Studio Code ou IntelliJ sont utilisés, cette vérification sera également effectuée sur ces derniers. La description de l'installation des plug-ins dans ceux-ci ne sera pas détaillée ici.

7.3.3 Émulateurs

Pour la dernière étape, qui vérifie la présence d'un appareil permettant l'affichage du rendu de nos applications par le biais d'un produit physique ou d'un émulateur, il existe plusieurs solutions.

La première peut consister à connecter un appareil directement à votre ordinateur via le câble d'origine fourni par le constructeur.

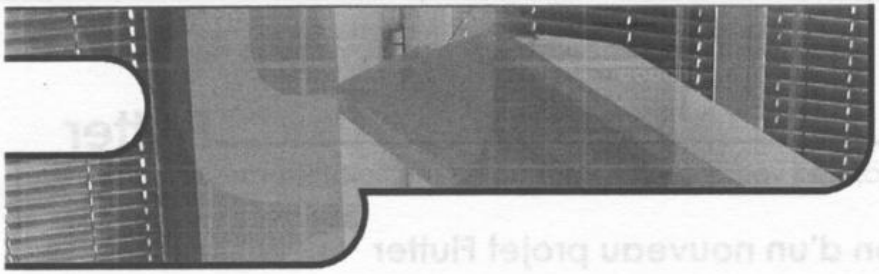
La seconde est d'utiliser un émulateur. Cela tombe bien puisqu'Android Studio va le fournir ainsi que Xcode pour ceux qui seraient sur Mac.

La configuration de l'émulateur intégré d'Android Studio sera abordée au chapitre suivant dans la section Les émulateurs.

En revanche, sur Mac il est possible de lancer tout de suite l'émulateur (sous réserve d'avoir installé Xcode) grâce à une ligne de commande :

```
% open -a simulator
```

Maintenant, il est possible de réexécuter la commande `flutter doctor`. Tous les voyants devraient être au vert hormis peut-être celui de l'émulateur si l'OS est Windows ou Linux. Dans tous les cas, il va être possible de lancer une première application et de procéder à la configuration de l'émulateur d'Android Studio.



Chapitre 3
Ma première application

1. Introduction

Après avoir terminé la configuration de l'environnement de travail, il ne reste plus qu'à lancer un premier projet Flutter.

Dans ce chapitre, l'enjeu sera de créer un nouveau projet Flutter. Pour y arriver, les étapes de création dans Android Studio seront détaillées. Sur ce nouveau projet, il sera alors possible de comprendre de quoi est constitué un projet Flutter à la base. D'abord, en examinant les fichiers qui le composent puis en regardant de plus près le code de cette première application. L'étape suivante consistera à la lancer dans l'émulateur. Ce sera également l'occasion d'évoquer le paramétrage de l'émulateur d'Android Studio. Enfin, il sera temps de produire les premières lignes de code pour réaliser une première application. Ce sera le moment idéal pour insérer un widget, élément central de Flutter. Tout ceci permettra de découvrir très succinctement à quoi peut ressembler la programmation avec Flutter, mais aussi d'apprécier le confort que procure le hot reload.

2. Création d'un nouveau projet Flutter

Pour cela, il faut bien sûr lancer Android Studio puis démarrer un nouveau projet Flutter.

En fonction de l'utilisation récente ou non de votre IDE, il existe deux possibilités. D'abord, s'il s'agit du premier lancement ou que tous les projets ont été clos, alors une boîte de dialogue telle que celle représentée à la figure 1 devrait apparaître. Dans ce cas, il suffira de choisir l'option **Start a new Flutter project** puis de poursuivre jusqu'à la section Choix du type de projet. Attention de ne pas confondre avec la première ligne qui propose de créer un nouveau projet Android classique.

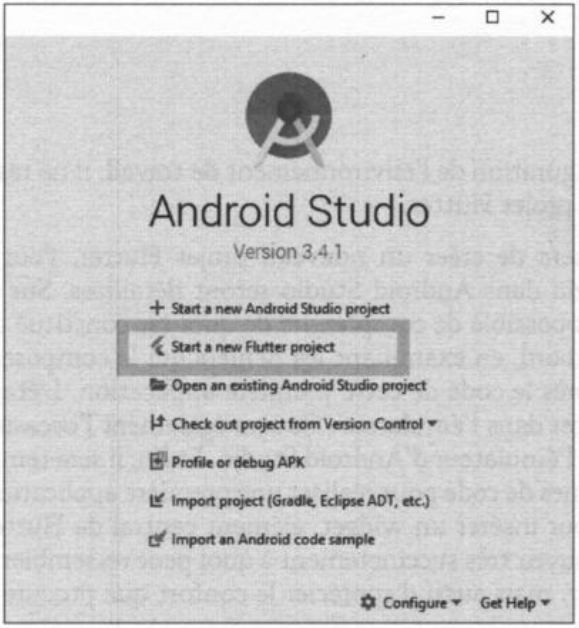


Figure 1 - Fenêtre de démarrage d'Android Studio

© Editions ENI - All rights reserved

Dans le second cas, si un projet est déjà ouvert, il faut soit le fermer en passant par **File** et **Close Project**, soit ouvrir directement un nouveau projet en cliquant sur **File**, **New** et enfin **New Flutter Project**. Le choix sera ensuite proposé d'ouvrir ce nouveau projet dans une nouvelle fenêtre ou dans la fenêtre actuelle.

En cas de fermeture d'un projet, Android Studio se retrouvera dans la configuration de la figure 1. Sinon, l'utilisateur sera amené à effectuer un choix sur le type de projet Flutter voulu (Figure 2).

2.1 Choix du type de projet

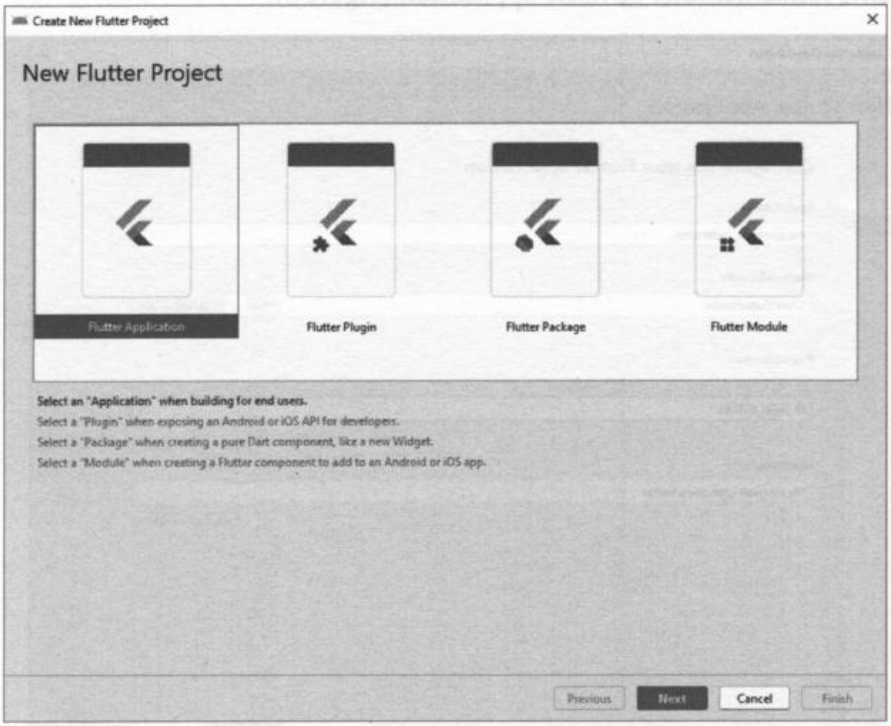


Figure 2 - Choix du type d'application

Dans le cas présent, afin de découvrir notre première application, il faudra choisir la première solution **Flutter Application**.

Les autres cas ont un nom suffisamment explicite pour comprendre leur utilité et l'ouvrage ne détaillera pas leur utilisation.

Après avoir cliqué sur **Next**, il sera temps de configurer cette première application.

2.2 Configuration de l'application

La boîte de dialogue présentée ci-dessous va permettre de renseigner des éléments fondamentaux de notre application (Figure 3).

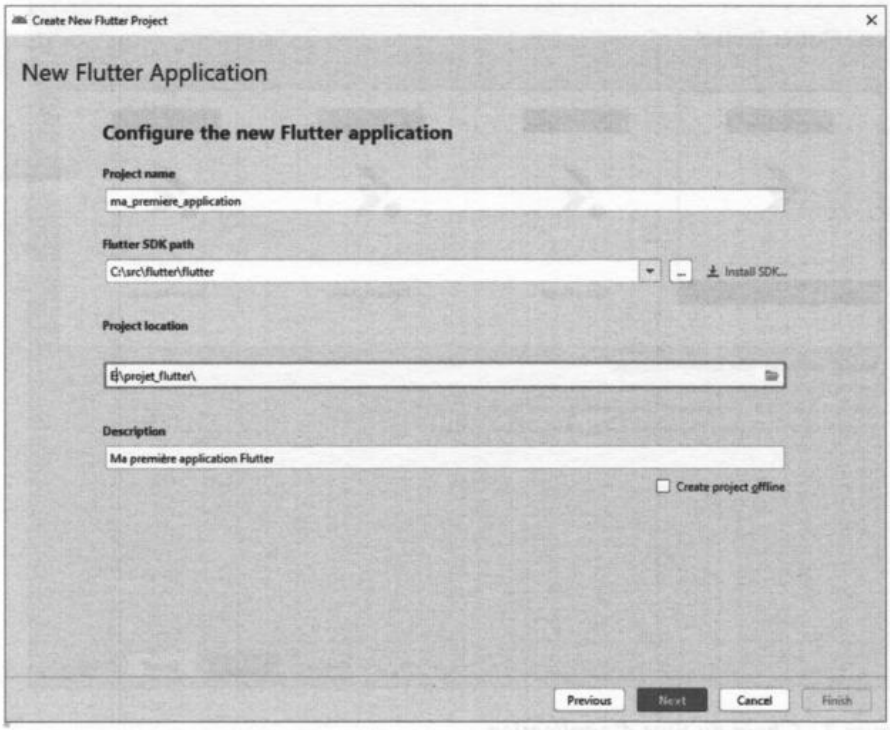


Figure 3 - Configuration de l'application

© Editions ENI - All rights reserved

Quatre champs sont à remplir :

- Pour commencer, il faut choisir un nom pour le projet.

Remarque

Attention au choix du nom. Celui-ci ne peut comporter que des lettres minuscules ainsi que le symbole du tiret bas (underscore en anglais).

- Le second champ est consacré à l'emplacement du SDK de Flutter. Normalement, suite aux étapes de configuration détaillées lors du chapitre Installation de l'environnement, elle devrait être renseignée. Toutefois, si elle ne l'était pas il faudra indiquer où se trouve le dossier du SDK. Pour cela, il suffit d'utiliser les trois points en bout de ligne (...).
- Le troisième élément à renseigner concerne l'emplacement où le projet va être enregistré. Il est conseillé d'éviter les dossiers nécessitant des droits élevés comme ceux de Windows, ainsi que ceux dont le nom comporte des espaces ou des caractères spéciaux.
- Enfin, dernier point, il est possible d'écrire une courte description du projet.

Une fois les quatre champs remplis, il suffit de cliquer sur **Next** pour passer à l'écran suivant.

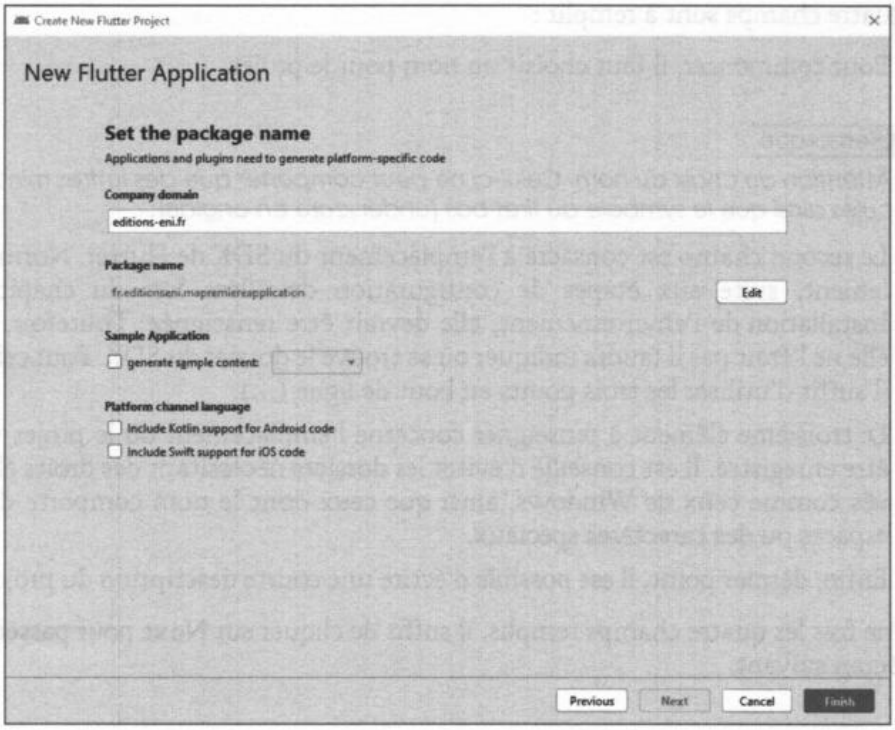


Figure 4 - Configuration de l'application 2

À cette étape, le nom de domaine de votre société (***Company domain***) est à compléter.

De quoi s'agit-il ?

Tout simplement, en entreprise ou dans une organisation, il convient d'indiquer le nom de domaine de celle-ci.

Dans l'exemple de la figure 4, le domaine est *editions-eni.fr*.

En tant que particulier, ou s'il n'y a aucun nom de domaine précis, il faudra tout de même renseigner cet élément.

© Editions ENI - All rights reserved

En effet, celui-ci sert à compléter le nom du package du projet. Le nom de domaine se retrouve écrit à l'envers et le nom du projet y est associé. C'est ce qui le rend unique. Toujours dans la figure 4, le nom du package (***Package name***) est, par conséquent, *fr.editionseni.mapremiereapplication*.

À ce niveau, il est également possible d'inclure du support pour Kotlin ou Swift si du développement spécifique pour Android ou iOS est prévu. Pour cette première application, cela ne sera pas nécessaire.

Il ne reste plus qu'à cliquer sur **Finish** pour lancer la génération du projet. Il faudra quelques instants pour qu'il se créé et voir apparaître le code déjà généré.

3. Le contenu de l'application

Après avoir cliqué sur **Finish** à l'étape précédente, un nouveau projet Flutter de base a été créé. Il s'agit d'une sorte de code de démonstration permettant d'apprécier quelques fonctionnalités de Flutter.

C'est l'occasion de s'attarder sur le contenu d'une application. Le but n'est pas d'être exhaustif, mais de brosser un tableau général permettant la compréhension de la structure d'un projet.

Quels sont les fichiers qui entrent en jeu ? Comment est constitué un projet Flutter ? Quel est le code de base d'une application ? À quoi ressemble le langage Dart ?

Voilà quelques questions qui vont trouver leur réponse dans cette partie.

3.1 L'architecture de l'application

Afin d'être compatible avec les plateformes Android et iOS, un projet Flutter intègre un certain nombre de fichiers qui seront bien connus des développeurs qui ont l'habitude de pratiquer sur ces OS. À cela, évidemment, viendront se rajouter des fichiers propres au SDK. La figure 5 donne un aperçu des fichiers qui composent globalement l'architecture d'un projet Flutter.

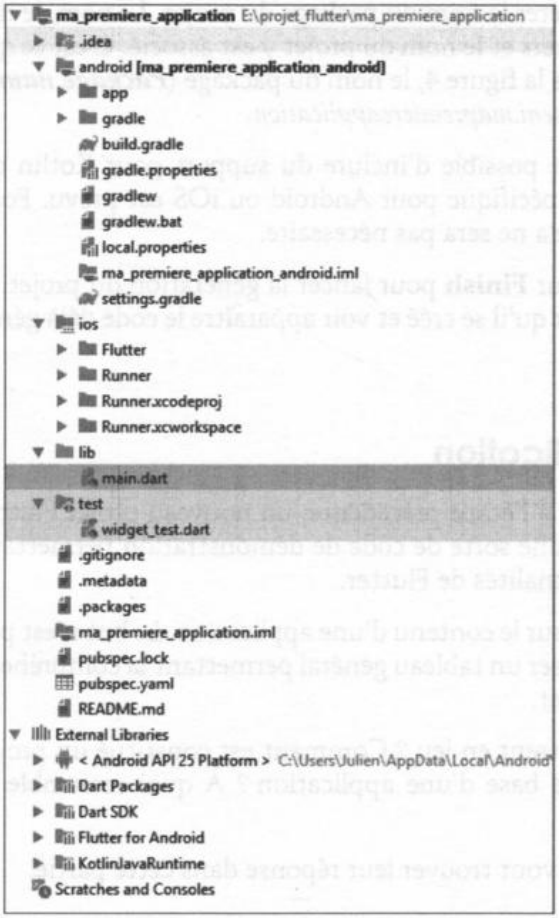


Figure 5 - Architecture de base Flutter

Un projet Flutter regroupe deux éléments principaux :

- **Le dossier du projet** (ici, *ma_premiere_application*) où tous les fichiers manipulables sont stockés. C'est la partie principale.
- **Les bibliothèques externes** (***External Libraries***), dossier qui contient les éléments utiles au bon fonctionnement du projet tels que les packages et SDK de Dart.

© Editions ENI - All rights reserved

Dans le premier dossier, celui du projet, il existe quatre dossiers importants ainsi que certains fichiers directement rattachés à la racine.

Les dossiers sont les suivants :

- **android** : tous les fichiers spécifiques à Android sont contenus ici. Il est possible de retrouver à l'intérieur les fichiers :
  - **gradle** : les fichiers Gradle sont des fichiers de configuration. De nombreuses informations sont contenues ici comme la version de l'API Android (*Application Programming Interface* ou interface de programmation d'application en français). La version de l'application y est également présente. Enfin, les bibliothèques utiles à l'application y sont déclarées.
  - **app** : Ce dossier contient le code de l'application Android. Dans le dossier ***src*** puis ***main*** est situé le fichier ***AndroidManifest.xml***. C'est le fichier auquel il faudra recourir pour attribuer des permissions spéciales à certaines fonctionnalités comme l'accès à Internet ou encore l'envoi de SMS. Par exemple, pour cette dernière demande, le code xml serait le suivant :

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

Pour les besoins du livre, il n'est pas utile de détailler davantage les différents fichiers qui se trouvent dans le dossier Android. Toutefois, pour ceux qui souhaitent aller plus loin et approfondir cet aspect, il existe un certain nombre de livres publiés aux Éditions ENI qui seront très complémentaires :

- *Android, Les fondamentaux du développement d'application Java* de Nazim BENBOURAHLA.
- *Android, guide du développement d'applications Java pour Smartphones et Tablettes* de Sylvain HEBUTERNE.
- *Android, Programmation en Java sous Android Studio* de Sylvain HEBUTERNE.
- *Kotlin, Les fondamentaux du développement d'applications Android* d'Anthony COSSON.

- **Ios** : tous les fichiers relatifs à iOS sont rangés à cet endroit. Le fichier AppDelegate.swift, par exemple, est présent. C'est le point d'entrée d'une application Swift. Le fichier Info.plist qui est une sorte d'équivalent à l'AndroidManifest.xml, se trouve ici également.

  À noter les livres sur le sujet aux Éditions ENI :

  Swift 3 pour iPhone, Développez vos premières applications pour iPhone de Pascal BATTY.

- **test** : ce dossier contient le fichier Dart widget_test.dart qui, comme son nom l'indique, est dédié aux tests. C'est dans ce dossier que l'ensemble du code servant à tester l'application devra se situer.
- **lib** : il s'agit du dossier qui va nous intéresser dans un premier temps. En effet, le fichier main.dart s'y trouve. Le code présenté dans la zone centrale au démarrage est celui de ce fichier. C'est le cœur de l'application. Plus tard, il sera possible de le découper afin de lui laisser uniquement la charge du lancement de l'application. Le reste du code, lui, sera dispersé logiquement dans d'autres fichiers.
- **pubspec.yaml** : Il s'agit d'un fichier de déclarations. Ici, les ressources (`assets`) telles que des images sont déclarées, tout comme les bibliothèques externes utiles au projet. Il s'agit d'un fichier majeur auquel il faudra recourir régulièrement.

  Ce fichier est aussi le lieu où les informations de base de l'application sont enregistrées. Son nom et sa description, qui ont été renseignés préalablement, se retrouvent donc là. La version de l'application ainsi que celles des différents SDK sont présentes.

Les autres fichiers seront décrits au fur et à mesure des besoins. Quoi qu'il en soit, dans le cadre des exercices et des démonstrations de cet ouvrage, les fichiers ou dossiers détaillés ci-dessus sont suffisants.

© Editions ENI - All rights reserved

3.2 Le code de l'application

Il est maintenant temps de se pencher plus attentivement sur le code qui a été généré à la création de ce premier projet.

C'est le moment de voir à quoi peut ressembler le langage Dart. Bien entendu, ses fondamentaux seront détaillés plus en profondeur ultérieurement.

Le code présenté ci-dessous est celui contenu dans le fichier main.dart du dossier lib.

L'analyse des deux premières lignes permet de voir qu'un système d'importation existe en Dart. Cet aspect sera revu lors du chapitre Dart, notions avancées.

La seconde ligne, `void main()`, est une fonction fléchée qui « pointe » vers une fonction nommée `runApp` qui, comme son nom l'indique, sert à spécifier quelle est la classe qui va être exécutée au démarrage de l'application. Ici, c'est la classe `MyApp` qui est concernée.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());
```

Cette classe se situe juste en dessous.

```
class MyApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       // This is the theme of your application.
       //
       // Try running your application with "flutter run". You'll see the
       // application has a blue toolbar. Then, without quitting the app, try
       // changing the primarySwatch below to Colors.green and then invoke
       // "hot reload" (press "r" in the console where you ran "flutter run",
       // or simply save your changes to "hot reload" in a Flutter IDE).
       // Notice that the counter didn't reset back to zero; the application
       // is not restarted.
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo Home Page'),
   );
 }
}
```

Cette première classe indique dans son nom `extends StatelessWidget`. Il s'agit d'une classe dont l'état n'évoluera pas au cours du temps (cf. chapitre Les états).

Il s'agit du point d'entrée de l'application. Quelques éléments importants y sont décrits comme le titre de l'application (`title`), le thème par défaut (`theme`) qui comporte, entre autres, la couleur primaire utilisée au niveau de `primarySwatch`. Le denier élément, le plus important peut-être, est la classe contenant le code décrivant l'affichage du premier écran (`home`). Elle porte le nom `MyHomePage` et prend en paramètre, entre les parenthèses, le titre qui sera affiché sur l'écran.

Elle comporte le code suivant :

```
class MyHomePage extends StatefulWidget {
MyHomePage({Key key, this.title}) : super(key: key);

  // This widget is the home page of your application. It is stateful, meaning
  // that it has a State object (defined below) that contains fields that affect
  // how it looks.

  // This class is the configuration for the state. It holds the values (in this
  // case the title) provided by the parent (in this case the App widget) and
  // used by the build method of the State. Fields in a Widget subclass are
  // always marked "final".

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

Contrairement à la première classe vue plus haut, ici `MyHomePage` hérite de `StatefulWidget`. Elle aura donc un comportement dynamique (cf. chapitre Les états).

Deux choses sont à noter. D'abord, le titre passé plus haut en paramètre est récupéré et associé à la ligne :

```
final String title;
```

La seconde, la plus importante, est la fonction `createState()` qui renvoie vers la classe `_MyHomePageState`.

© Editions ENI - All rights reserved

Il y a donc un dernier renvoi avant d'accéder au code qui décrit l'affichage :

```
class _MyHomePageState extends State<MyHomePage> {
 int _counter = 0;

 void _incrementCounter() {
   setState(() {
     // This call to setState tells the Flutter framework that something has
     // changed in this State, which causes it to rerun the build method below
     // so that the display can reflect the updated values. If we changed
     // _counter without calling setState(), then the build method would not be
     // called again, and so nothing would appear to happen.
     _counter++;
   });
 }

 @override
 Widget build(BuildContext context) {
   // This method is rerun every time setState is called, for instance as done
   // by the _incrementCounter method above.
   //
   // The Flutter framework has been optimized to make rerunning build methods
   // fast, so that you can just rebuild anything that needs updating rather
   // than having to individually change instances of widgets.
   return Scaffold(
     appBar: AppBar(
       // Here we take the value from the MyHomePage object that was created by
       // the App.build method, and use it to set our appbar title.
       title: Text(widget.title),
     ),
     body: Center(
       // Center is a layout widget. It takes a single child and positions it
       // in the middle of the parent.
       child: Column(
         // Column is also a layout widget. It takes a list of children and
         // arranges them vertically. By default, it sizes itself to fit its
         // children horizontally, and tries to be as tall as its parent.
         //
         // Invoke "debug painting" (press "p" in the console, choose the
         // "Toggle Debug Paint" action from the Flutter Inspector in Android
         // Studio, or the "Toggle Debug Paint" command in Visual Studio Code)
         // to see the wireframe for each widget.
         //
         // Column has various properties to control how it sizes itself and
         // how it positions its children. Here we use mainAxisAlignment to
         // center the children vertically; the main axis here is the vertical
         // axis because Columns are vertical (the cross axis would be
         // horizontal).
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             'You have pushed the button this many times:',
           ),
           Text(
             '$_counter',
             style: Theme.of(context).textTheme.display1,
           ),
         ],
```

```
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  );
}
}
```

Cette classe contient plus d'informations. Pour commencer, une variable portant le nom de `_counter` est initialisée avec la valeur 0. Il s'agit d'un compteur qui va enregistrer le nombre de fois où un bouton sera utilisé.

Juste en dessous, une procédure (`void _incrementCounter()`) contient une méthode nommée `setState()` qui va se charger de faire évoluer dynamiquement la valeur de la variable `_counter` en l'incrémentant de 1 à chaque fois qu'elle sera appelée. Cette notion sera revue en détail au cours du chapitre Les états.

Le troisième bloc de cette classe décrit enfin le premier widget de l'application. Pour rappel, avec Flutter tout est widget. Ils sont les composants de base de l'application. La plupart correspondent à des éléments graphiques qui ont été recréés pour copier le style des OS (Android et iOS). C'est le cas des boutons, des barres de menu. Certains autres ont un rôle plus complexe et pas nécessairement lié à l'affichage. Les widgets feront l'objet de plusieurs chapitres par la suite.

Le premier widget ici nommé `build` est, comme son nom l'indique, la fondation ou la construction de l'écran. Il retourne dans l'exemple un autre widget, `Scaffold`, qui correspond à la structure classique d'une application Android de type **Material Design**.

© Editions ENI - All rights reserved

Dans le code présenté, le `Scaffold` est composé de trois éléments :

- Une `appBar` qui représente la barre supérieure colorée d'un écran. Le titre de ce dernier y est inscrit.
- Un `body` qui sera la partie centrale de l'écran. À l'intérieur se situe encore un widget : `Center`. Ce dernier sert à centrer les widgets que lui-même contient. Ici, il ne possède qu'un seul et unique enfant (`child`), `Column`, qui traduit en français désigne une colonne. À l'évidence, les widgets qui y seront intégrés seront disposés selon ce schéma. D'ailleurs, on s'attend à avoir plusieurs éléments sur une colonne, et c'est le cas puisque pour les référencer il existe un tableau de widgets (`children : <Widget> [...]`). L'autre caractéristique de la colonne est de pouvoir préciser la disposition ou l'alignement grâce à `mainAxisAlignment`. Concernant les enfants situés dans cette colonne, il en existe deux. Il s'agit de textes (`Text`). Le premier contient juste une phrase (*« You have pushed the button this many times : »*) qui dit, en français : « Vous avez appuyé sur le bouton ce nombre de fois : ». L'autre affiche juste la valeur de la variable `_counter`. À cette fin, la variable est précédée du symbole `$`.
- Un `floatingActionButton`, ou traduit en français, un bouton d'action flottant qui, dans l'exemple, est relié au compteur `_counter`. Il contient trois éléments :
  - `onPressed` : indique quelle méthode sera appelée suite à un appui sur le bouton. Dans l'exemple, l'appel se fait sur `_incrementCounter` détaillée un peu plus haut.
  - `tooltip` : il correspond à un message passant présenté dans une bulle d'information et déclenché lors d'un appui long sur le bouton.
  - `child` : ce widget possède lui aussi un enfant. Bien évidemment, il s'agit aussi d'un widget. Une icone en forme de « + » ici.

Après avoir analysé rapidement ce code, il est maintenant temps d'obtenir un rendu dans un émulateur.

4. Les émulateurs

4.1 Introduction

Les émulateurs sont un élément essentiel dans la réalisation d'une application. Ils permettent de tester et d'avoir un rendu fiable sur une large gamme de produits.

L'émulateur Android est déjà intégré à Android Studio. En revanche, pour en avoir un sur iOS, afin d'effectuer un rendu dans le cadre d'un développement cross-platform par exemple, il faut remplir l'unique condition qui est d'être sur macOS. En effet, au travers de Xcode (indisponible en dehors de macOS), dont l'installation a été détaillée dans le chapitre précédent, il sera possible de lancer un émulateur de l'iPhone.

S'attarder sur les émulateurs est aussi l'occasion de reparler d'un des avantages non négligeables de Flutter. Pendant l'émulation, pour revenir sur ce qui a été décrit dans la seconde partie du chapitre Introduction, le développeur pourra s'appuyer sur le hot reload. Pour rappel, ce mécanisme permet de voir toutes les modifications en direct sur l'émulateur sans avoir à relancer la compilation du programme. Il procure un gain de temps incroyable lors des phases de développement. L'autre avantage est que l'état de l'application reste le même. C'est-à-dire que les valeurs des différentes variables restent inchangées. De même, la navigation réalisée d'écran en écran n'est pas remise en cause. Elle n'est pas réinitialisée, il n'y a donc pas besoin de reparcourir plusieurs écrans pour revenir sur celui qui fait l'objet d'une modification.

Comment activer cet outil ? Il n'y a rien à faire, il est déjà inclus. Il suffira de bien penser à sauvegarder à la suite d'une modification du code pour obtenir le résultat immédiatement.

Maintenant que ces quelques explications ont été données, il convient de finaliser la dernière étape de l'installation, la configuration de l'émulateur Android.

© Editions ENI - All rights reserved

4.2 Paramétrage de l'émulateur Android

À la fin du chapitre précédent, il restait une étape importante qui n'avait pas été effectuée : la configuration de l'émulateur Android. Pour que cette étape soit facilement réalisée, il convient d'avoir un projet ouvert sur Android Studio. Cela tombe bien puisqu'avec ce premier projet Flutter, cette condition est remplie.

Pour procéder au paramétrage de l'émulateur, il faut se rendre dans la barre de menu sur l'onglet **Tools** puis cliquer sur **AVD Manager**.

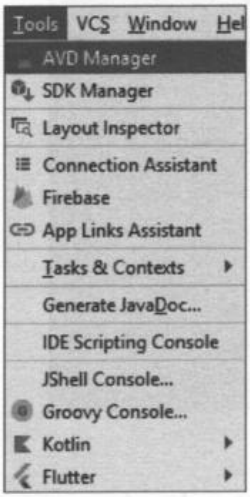


Figure 6 - Menu AVD Manager

Remarque

*Avant de passer à la suite, il est intéressant de constater que la dernière ligne de **Tools** est **Flutter**. En passant par ici, Android Studio donne accès aux lignes de commande de Flutter comme **flutter doctor** par exemple. Il n'est donc plus nécessaire d'avoir un terminal ouvert en parallèle, tout peut se faire depuis l'IDE.*

La fenêtre suivante va s'ouvrir, il s'agit de l'*Android Virtual Device Manager* (***AVD Manager***) :

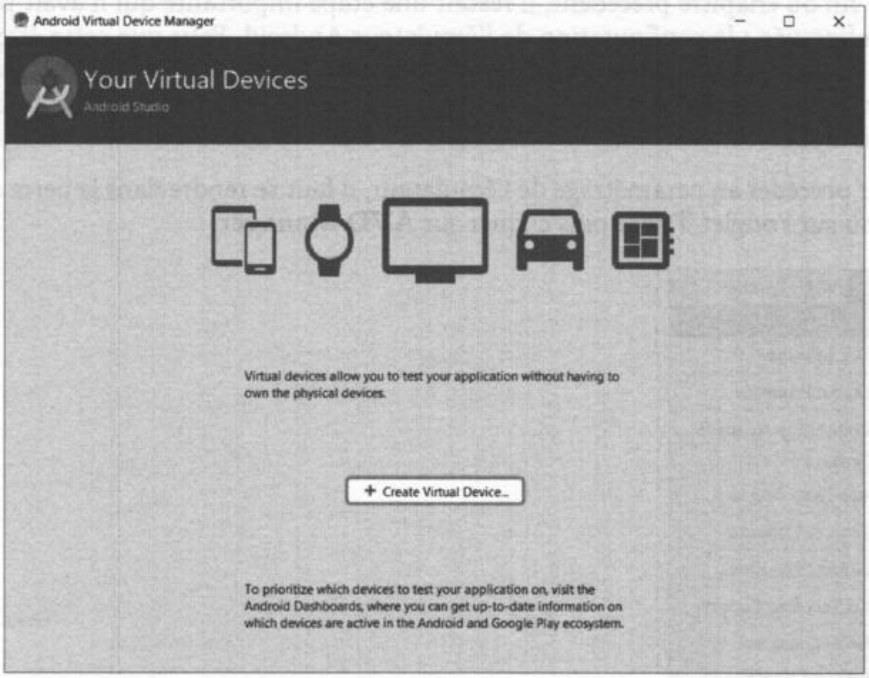


Figure 7 - Android Virtual Devices Manager

L'AVD Manager est un outil de configuration qui permet de créer tout type d'émulateur Android pouvant aller du smartphone à la tablette, en passant par le système embarqué (TV) ou encore la montre connectée.

Sur l'écran de présentation illustré par la figure 7, il convient de cliquer sur **+ Create Virtual Device...**.

© Editions ENI - All rights reserved

Une nouvelle fenêtre prend le relais :

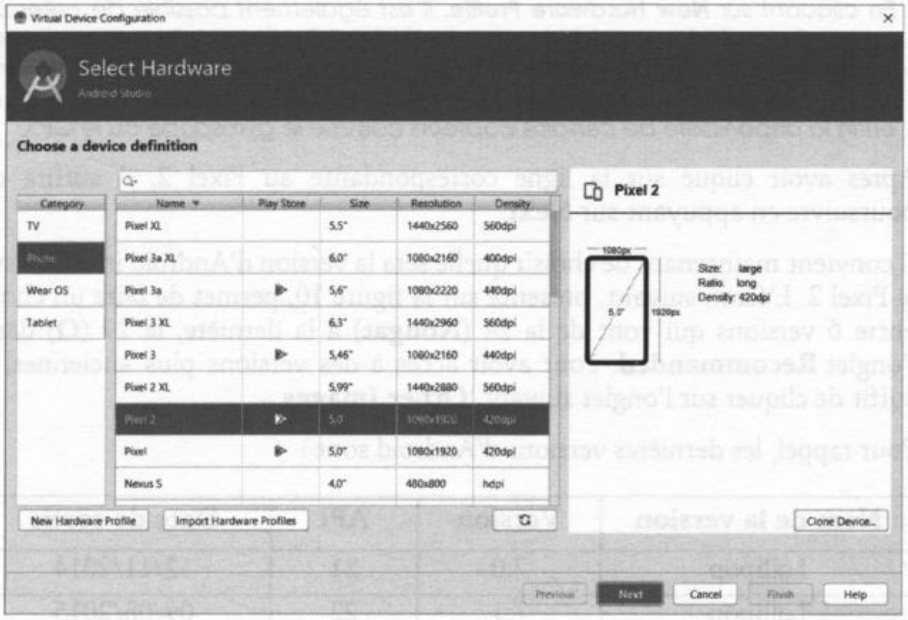


Figure 8 - Choix de l'appareil émulé

Sur cet écran, l'enjeu est de sélectionner un émulateur de smartphone assez commun. En soi, un appareil que l'on pourrait retrouver chez une multitude d'utilisateurs.

Pour commencer, sur la partie de gauche, il faut penser à choisir **Phone**. Normalement, ce choix est présenté par défaut.

Sur la partie centrale, ensuite, une gamme de produits est proposée. Pour les démonstrations, le choix va se porter le Pixel 2. Sa taille d'écran et sa résolution sont assez standards.

Remarque

*En cliquant sur **New Hardware Profile**, il est également possible de créer un appareil qui n'est pas contenu dans cette liste. Quelques informations seront demandées comme le nom du modèle, son type, la taille de son écran ainsi que sa résolution, la quantité de RAM, la présence éventuelle de caméra, et enfin la disponibilité de certains capteurs comme le gyroscope ou le GPS.*

Après avoir cliqué sur la ligne correspondante au Pixel 2, il suffira de poursuivre en appuyant sur **Next**.

Il convient maintenant de choisir quelle sera la version d'Android installée sur le Pixel 2. L'écran suivant, présenté sur la figure 10, permet de faire un choix entre 6 versions qui vont de la 24 (**Nougat**) à la dernière, la 29 (**Q**) dans l'onglet **Recommended**. Pour avoir accès à des versions plus anciennes, il suffit de cliquer sur l'onglet suivant, **Other Images**.

Pour rappel, les dernières versions d'Android sont :

Nom de la version	Version	API	Date de sortie
Lollipop	7.0	21	12/11/2014
Lollipop	7.1	22	09/03/2015
Marshmallow	6.0	23	05/10/2015
Nougat	7.0	24	22/08/2016
Nougat	7.1	25	04/10/2016
Oreo	8.0	26	21/08/2017
Oreo	8.1	27	05/12/2017
Pie	9.0	28	06/08/2018
Android 10 (Q)	10.0	29	03/09/2019

Pour savoir quelle version choisir, Google met à disposition un graphique montrant le pourcentage d'utilisateurs sur chacune d'elles.

© Editions ENI - All rights reserved

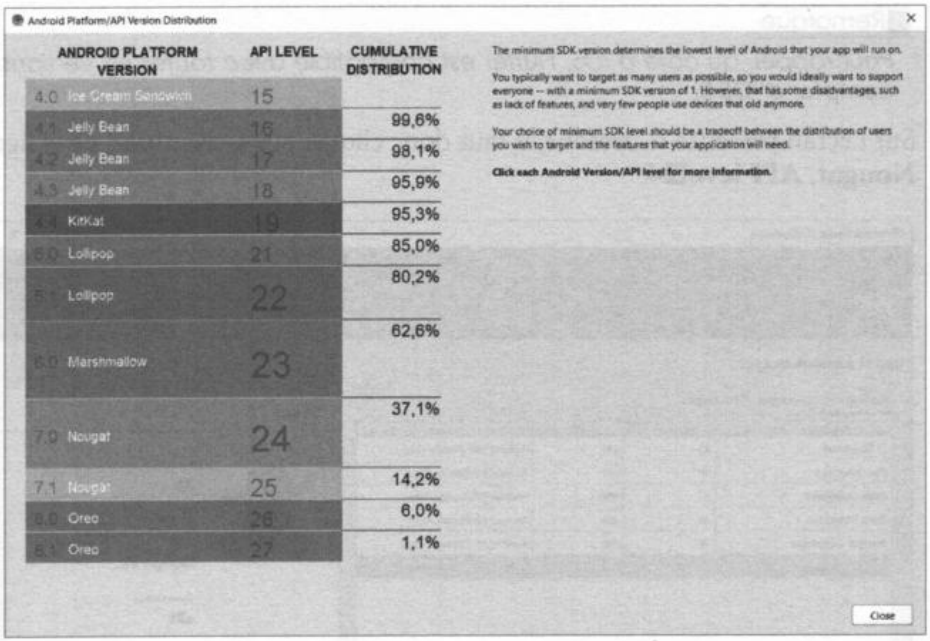


Figure 9 - Pourcentage d'utilisateurs par version

Sur la figure 9, il est indiqué pour chaque plateforme le pourcentage cumulé d'utilisateurs. Par exemple, il y a 85 % des utilisateurs qui ont au moins la version 5.0 d'Android (ou supérieure). Ce qui suppose que 15 % se servent d'une version plus ancienne.

Dans le choix de la version préinstallée sur l'émulateur, il n'y aura pas d'impact important pour l'instant. Mais plus tard, il pourrait être intéressant d'observer le comportement des applications sur différentes versions.

Dans le cadre de cet ouvrage, afin d'uniformiser le rendu, le Pixel 2 sera équipé de l'API 24, c'est-à-dire la version 7.0 appelée **Nougat**.

Remarque

*Flutter est compatible à partir de **Jelly Bean 4.1**, API 16 sortie le 9 juillet 2012. Selon, les statistiques, les utilisateurs d'Android se servant de cette version ou d'une plus récente représentent 99,6 %. Soit la quasi-totalité.*

Remarque

Pour rappel, du côté d'iOS, Flutter est compatible avec toutes les versions à compter d'iOS 8.

Sur l'écran présenté ci-dessous, il faut donc cliquer sur **Download** de la ligne **Nougat**, **API level24**.

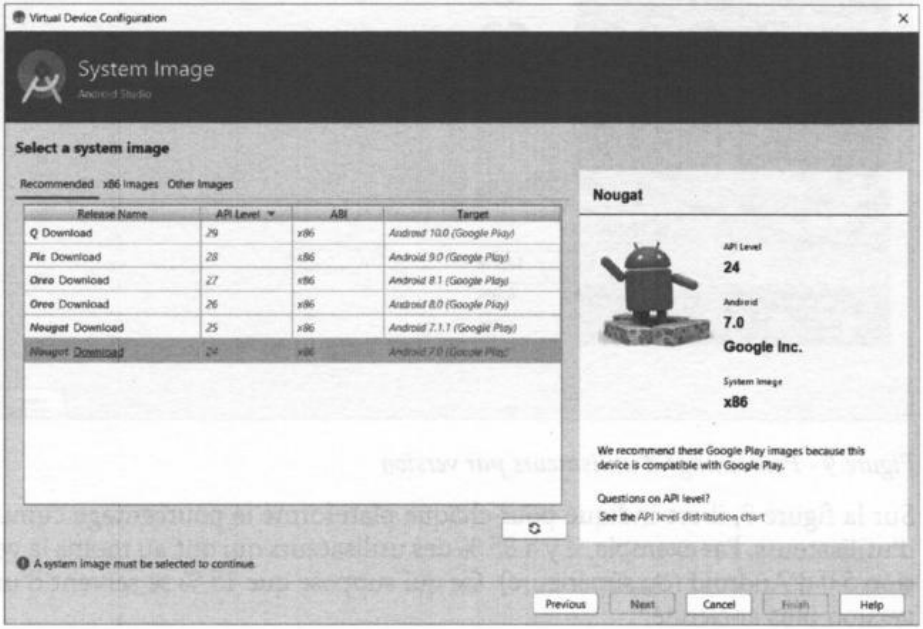


Figure 10 - Choix de la version Android de l'émulateur

Le téléchargement et l'installation peuvent s'avérer assez longs. En effet, l'image d'Android **Nougat** pèse plus de 775 Mo. Ceux-ci terminés, il sera possible de cliquer sur **Finish**.

La fenêtre se fermera pour revenir sur l'écran présenté sur la figure 10. La seule différence est que, maintenant, la mention « Download » de la version Nougat choisie aura disparu.

Si ce n'est pas déjà le cas, il faut resélectionner la ligne correspondant à la version téléchargée. Il ne reste plus qu'à cliquer sur **Next**.

© Editions ENI - All rights reserved

L'écran final va permettre de renseigner les derniers éléments de l'émulateur (cf. figure 11).

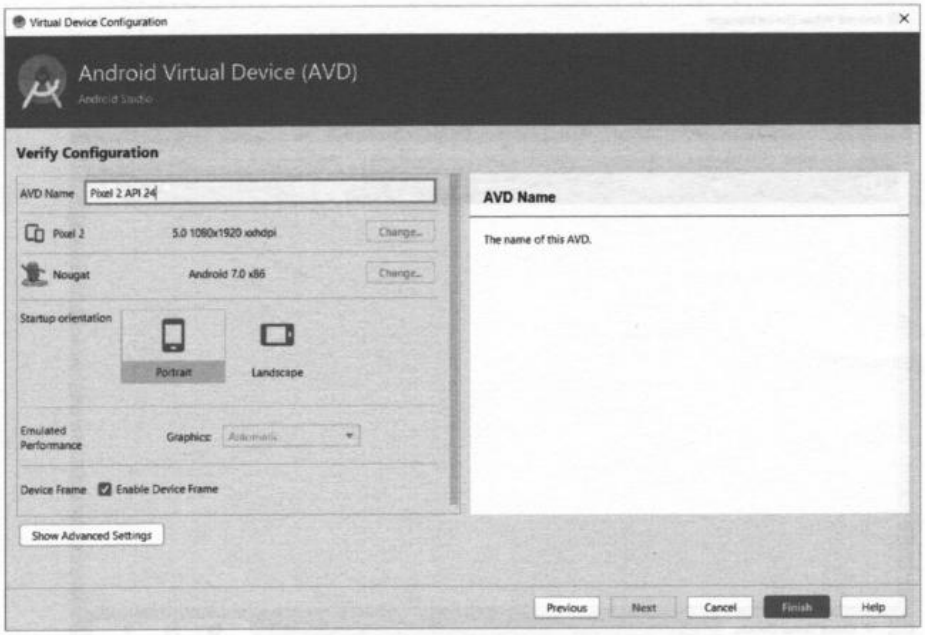


Figure 11 - Configuration de l'émulateur

À cette étape, il est possible de personnaliser le nom de l'émulateur. Une autre option permet de choisir l'orientation de l'appareil au démarrage soit en portrait, soit en paysage (**Landscape**). Il faut savoir qu'une fois en service, il est possible de « tourner » l'émulateur pour voir le rendu dans les deux cas d'orientation.

Une autre option, qui peut se révéler utile, concerne les performances de l'émulateur (**Emulated Performance**) qui peuvent être améliorées en fonction de la présence d'une carte graphique dédiée dans la machine hôte. Il faudra donc choisir **Hardware** si une carte graphique est présente et **Software** dans le cas contraire. En cas de doute, le système va pouvoir se débrouiller grâce au choix **Automatic**.

Dans notre situation, il convient de laisser les paramètres par défaut qui feront parfaitement l'affaire et de cliquer sur **Finish**.

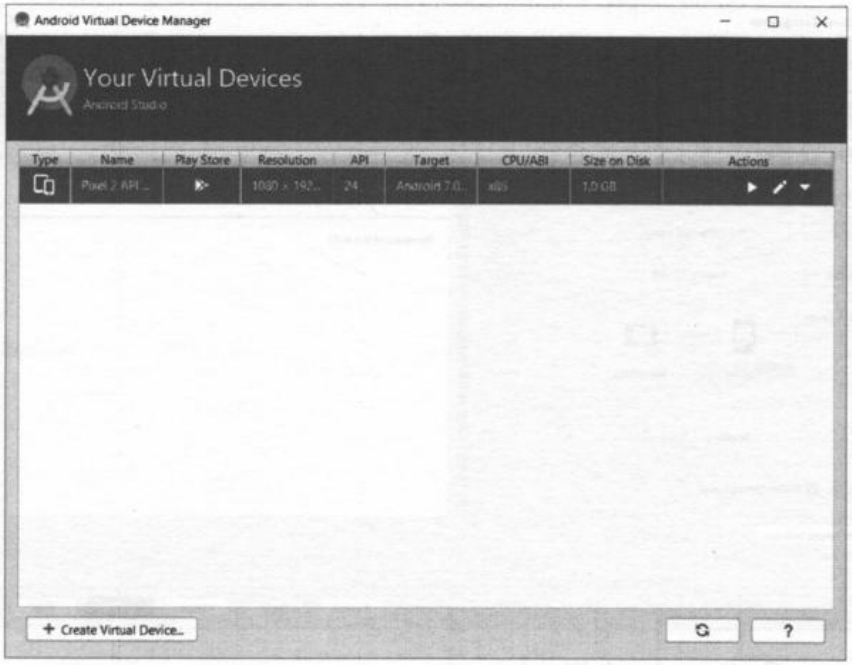


Figure 12 - Liste des émulateurs actifs

Sur la nouvelle fenêtre qui s'ouvre (Figure 12), Android Studio dresse la liste des émulateurs disponibles. Dans le cas présent, le Pixel 2 avec sa version 7.0 d'Android est bien là. Sur la droite, en bout de ligne, trois icônes offrent des options pour l'émulateur. D'abord, un bouton lecture ( ) qui permet de lancer l'émulation. Au centre, le stylo ( ) donne accès à l'édition et donc à la modification des paramètres de l'émulateur. Enfin, la flèche vers le bas ( ) ouvre un menu déroulant offrant quelques options supplémentaires.

À notre niveau, il est temps de tester cet émulateur Pixel 2 en appuyant sur le bouton de lecture ( ).

© Editions ENI - All rights reserved

Au bout de quelques secondes, il devrait apparaître :

Figure 13 - Emulateur Pixel 2 sous Android 7.0

L'émulateur va ensuite démarrer Android. Cette étape peut prendre un peu de temps comme sur un portable physique. Une fois l'écran principal atteint, l'application va pouvoir être lancée.

4.3 Lancement de la première application

Maintenant que l'émulateur est en cours d'exécution, il suffit de retourner sur Android Studio et de cliquer sur ▶ (**run**) placé en haut à droite.

Figure 14 - Barre de lancement

Après avoir cliqué sur le bouton **run**, il faut patienter un peu. Le premier lancement est assez long, en raison principalement de l'initialisation de Gradle.

Au travers de toutes les étapes qui vont s'exécuter, il est même probable, sur Windows, qu'une demande d'autorisation s'ouvre comme celle illustrée sur la figure 14.

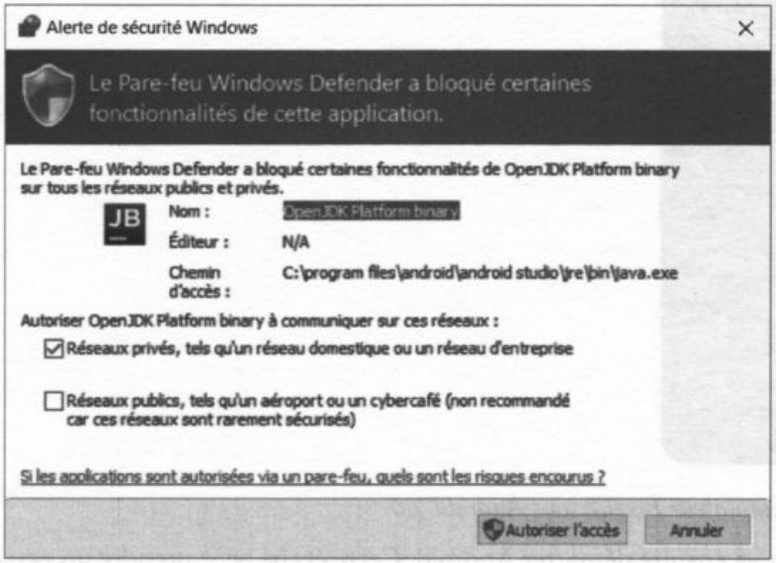


Figure 14 - Autorisation du Pare-feu Windows Defender

© Editions ENI - All rights reserved

Avec un peu de patience, le sésame apparaît enfin dans l'émulateur. Le rendu de l'application est le suivant :

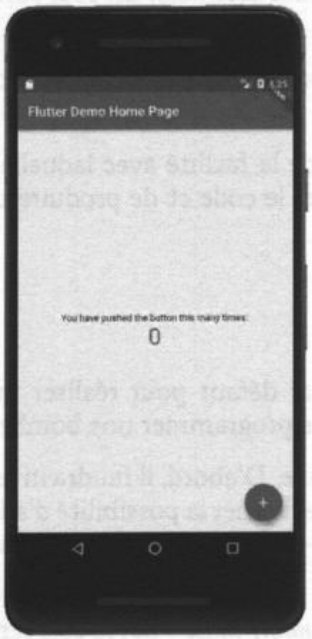


Figure 15 - Rendu de l'application

En lisant le code, il était possible de connaitre à l'avance les fonctionnalités de l'application. Mais avec le visuel, c'est encore plus évident.

Pour revenir sur les différentes parties, nous pouvons retrouver, située en haut, l'`AppBar` qui contient le titre « *Flutter Demo Home Page* ». Elle est de couleur bleue puisque c'est la valeur donnée à `primarySwatch`.

Au milieu, les deux textes sont bien centrés et l'un en dessous de l'autre. Cet effet est dû respectivement aux widgets `Center` et `Column`.

Enfin, en bas à droite, le bouton flottant attend un appui.

Cette action va entraîner la mise à jour immédiate du compteur. Le « *0* » affiché sur la deuxième ligne de texte va donc passer à « *1* » et ainsi de suite.

Il est intéressant de remarquer qu'avec très peu de code il est aisé de construire une application capable de gérer des évènements. Bien sûr, elle est compatible avec Android et iOS. Là où il aurait fallu deux codes distincts, un seul suffit désormais.

Il faut noter aussi la fluidité de l'application. Lors de l'appui sur le bouton flottant, aucune latence n'est perceptible.

Maintenant, afin de vraiment se rendre compte de la facilité avec laquelle il est permis de programmer, il convient de modifier le code et de produire un premier widget.

5. Premiers widgets

L'exercice va être de modifier le code proposé par défaut pour réaliser une application un peu différente. L'objectif fixé est de programmer une bombe !

Pour cela, il faut convenir de plusieurs points d'étape. D'abord, il faudra inverser la marche du compteur pour démarrer à 10 et se donner la possibilité d'aller jusqu'à 0. Cela suppose également que l'icône « + » du bouton flottant fasse l'objet d'une modification.

Rendu à 0, le texte « *You have pushed the button this many times* » devra être remplacé par un joli « *BOUM* ». Il conviendra aussi de modifier le style de la police pour qu'à ce moment le texte apparaisse en rouge et soit plus gros.

Au moment de l'explosion, il est normal que le fond de l'écran change de couleur et devienne orange.

Enfin, le compteur sera bloqué à 0 et il ne sera pas possible d'atteindre des valeurs négatives.

Pour mettre en route une bombe, il faut avant tout un compte à rebours. L'idée est ici de réutiliser la variable _counter et le bouton flottant pour donner cet effet.

Dans la classe `_MyHomePageState`, la valeur de _counter doit être de 10.

```
int _counter = 10;
```

© Editions ENI - All rights reserved

Puis pour décrémenter, il faut modifier le code contenu dans `setState()` :

```
_counter--;
```

Pour que la démarche soit logique, il faut retoucher au look du bouton flottant. L'icône doit être changée ainsi que le Tooltip.

```
floatingActionButton: FloatingActionButton(
 onPressed: _decrementCounter,
 tooltip: 'Compte à rebours',
 child: Icon(Icons.history),
),
```

Enfin, pour être totalement cohérent, le nom de la méthode appelée `_incrementCounter` doit être modifié pour `_decrementCounter`. La valeur du `tooltip` est modifiée pour « *Compte à rebours* » tandis que l'icône devient `Icons.history` ou plus explicitement le symbole d'une montre que l'on remonte à l'envers.

Avec la modification de `_incrementCounter`, une erreur est levée. En effet, nous indiquons que la méthode `_decrementCounter` devra être appelée dès que le bouton flottant sera pressé (`onPressed`), or il n'y a nulle part dans le code une méthode portant ce nom. Il faut modifier le nom de la méthode `_incrementCounter` située plus haut. Elle devient donc :

```
void _decrementCounter() {
```

Maintenant que des premières modifications visuelles ont été apportées et qu'il n'y a plus d'erreur, il est temps de voir à l'œuvre le hot reload. Pour cela il suffit de sauvegarder. Sur le Pixel 2 émulé, le bouton flottant a changé d'aspect.

L'idée est ensuite de faire en sorte que la première ligne de Text correspondant à « *You have pushed the button this many times* » change une fois le compteur arrivé à 0. Mais avant cela, il va falloir modifier la phrase. Elle n'a plus sa place ici dans un contexte de compte à rebours.

Pour cette manipulation, le plus simple va être de créer une nouvelle variable appelée `_phrase`. Elle pourra ensuite évoluer en fonction des besoins. Elle sera rajoutée juste en dessous de la variable `_counter` déjà présente au début de la classe `_MyHomePageState` :

```
int _counter = 10;
String _phrase = "Clic avant l'explosion :";
```

Il faut utiliser cette variable pour qu'elle soit affichée sur l'écran. La première zone de texte qui était :

```
Text(
  'You have pushed the button this many times:',
),
```

Devient donc :

```
Text(
  '$_phrase',
),
```

L'étape suivante doit modifier le contenu de `_phrase` au moment même où le compteur arrive à 0. À ce moment-là, le texte sera « BOUM » pour simuler l'explosion. Afin de réaliser cela, il faut utiliser une zone « d'observation ». Cela tombe bien, il en existe justement une avec `setState()`.

Il faut donc y rajouter ce code :

```
if (_counter == 0) {
  _phrase = "BOUM";
};
```

Il s'agit d'un bon début, mais, il serait encore mieux de voir le graphisme de ce texte évoluer aussi. Pour cela, il va falloir créer un nouveau widget `TextStyle` qui, comme son nom l'indique, a vocation à donner un style particulier à du texte.

Au début de la classe `_MyHomePageState`, il faut rajouter ce widget :

```
TextStyle _tS = TextStyle(
  color: Colors.green,
  fontSize: 15,
);
```

© Editions ENI - All rights reserved

Ici, le but est de créer un style pour le texte de la première ligne. À l'initialisation, le code ci-dessus sera chargé et la phrase « Clic avant explosion : » sera donc écrite en vert avec une taille de police de 15.

Pour que code soit actif, il convient de l'associer à la bonne zone de texte :

```
Text(
  '$_phrase',
  style: _tS,
),
```

Le widget `Text` accepte de recevoir un `style`. Il est donc possible de lui transmettre le `TextStyle _tS` créé précédemment.

Il reste encore une dernière étape pour rendre le tout parfaitement fonctionnel. Ce style doit évoluer : quand le compteur sera arrivé à 0, le texte doit devenir rouge, avec une taille de police plus élevée et en gras. C'est encore une fois dans `setState()` que cela va se jouer :

```
if (_counter == 0) {
  _phrase = "BOUM";
  _tS = TextStyle(
    color: Colors.red,
    fontSize: 30,
    fontWeight: FontWeight.bold,
  );
}
```

Pour renforcer l'effet d'explosion, il reste encore une petite chose à intégrer. Dans l'idéal, le fond d'écran qui est blanc pour l'instant devrait devenir orangé pour évoquer des flammes ou la chaleur.

Un nouveau widget est nécessaire. Il s'agit de `Color`. Encore une fois, il faut le déclarer au début de la classe `_MyHomePageState`.

```
Color _couleur = Colors.white;
```

Le code ci-dessus est un widget `Color` nommé `_couleur` et défini sur du blanc, comme l'écran actuel. Il faut maintenant l'associer à un autre widget. Lequel ?

Dans nos éléments affichés, celui sur lequel il sera logique de s'appuyer est body. Pour rappel, il s'agit de la zone centrale de l'application. C'est d'ailleurs un terme qui devrait parler aux développeurs ayant déjà pratiqué le HTML.

En fouillant un peu, il est facile de découvrir que body possède une propriété intitulée backgroundColor. Il est alors envisageable d'écrire cette ligne juste après la parenthèse fermante de Center en n'omettant pas d'ajouter une virgule.

```
), //Fin de Center
backgroundColor: _couleur,
```

De nouveau, setState() va gérer la modification de comportement. Avec le rajout de la couleur, le code devrait être le suivant :

```
if (_counter == 0) {
 _phrase = "BOUM";
 _tS = TextStyle(
   color: Colors.red,
   fontSize: 30,
   fontWeight: FontWeight.bold,
 );
 _couleur = Colors.amber;
}
```

L'évolution de _couleur se porte sur la couleur amber qui correspond à un orange assez soutenu.

Il reste une ultime étape à programmer. Pour être cohérent, le compteur ne doit pas pouvoir aller en deçà de 0. Il faut faire en sorte de le bloquer une fois qu'il aura atteint cette valeur. Encore une fois, cette fonctionnalité va se trouver dans setState(). Pour procéder, il convient d'ajouter une condition qui va venir entourer la décrémentation du compteur déjà présente :

```
if (_counter > 0) {
 _counter--;
}
```

© Editions ENI - All rights reserved

Avant d'afficher le rendu final de l'application ainsi que le code récapitulatif, certains auront peut-être remarqué la présence d'une petite bannière rouge avec l'inscription « DEBUG » en haut à droite de l'écran de l'émulateur. Cette indication, utile pendant les phases de développement, peut, dans la mesure où cette application est maintenant terminée, être retirée.

Figure 16 - Bannière de Debug

Pour cela, il faut saisir la ligne de code suivante dans la classe `MyApp` :

```
debugShowCheckedModeBanner: false,
```

Afin de voir ses effets, il ne suffit plus d'enregistrer. Le hot reload ne pourra pas fonctionner sur des propriétés de ce type. Il suffit de cliquer sur le triangle vert, correspondant au « Run », qui va faire redémarrer l'application. Cette manœuvre prendra moins longtemps que la première fois où l'application a été lancée puisqu'un certain nombre d'éléments n'auront pas besoin d'être rechargés.

Il est temps de voir à quoi ressemble cette application. D'abord au démarrage :

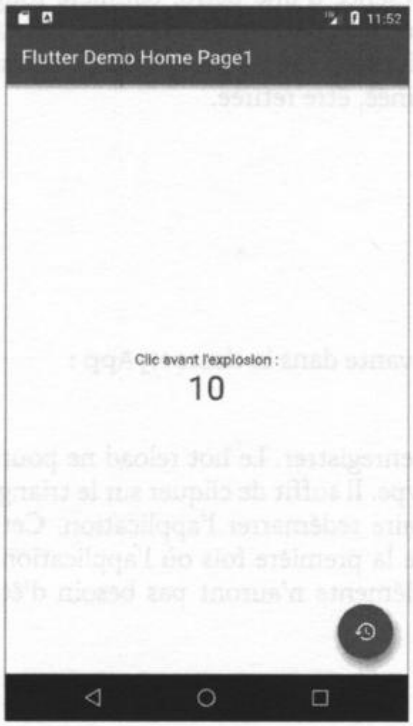


Figure 17 - Application au démarrage

© Editions ENI - All rights reserved

Après avoir appuyé sur le bouton flottant le compteur va commencer à diminuer. Puis, au moment où il va atteindre 0, l'écran va se changer pour devenir celui-ci :

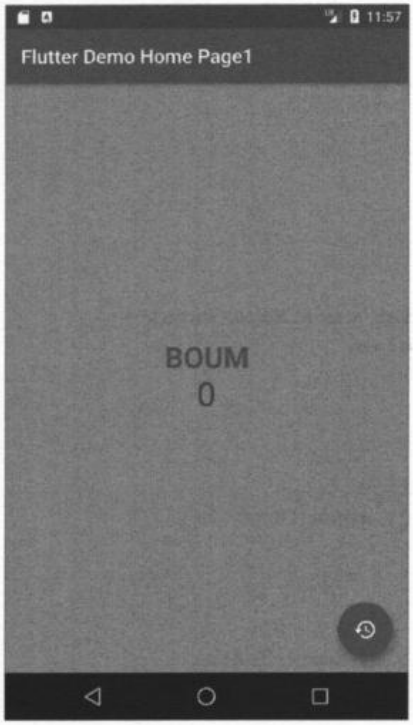


Figure 18 - Application au moment où le compteur atteint 0

L'explosion vient d'avoir lieu. Si l'utilisateur essaie encore d'appuyer sur le bouton flottant, plus rien ne se passe. Pour relancer l'application, il faut cliquer sur le triangle vert (**Run**).

Afin de récapituler, et plus simplement au cas où les explications ci-dessus n'auraient pas été claires, voici le code complet de l'application :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo1',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo Home Page1'),
     debugShowCheckedModeBanner: false,
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
 int _counter = 10;
 String _phrase = "Clic avant l'explosion :";
 TextStyle _tS = TextStyle(
   color: Colors.green,
   fontSize: 15,
 );
 Color _couleur = Colors.white;

 void _decrementCounter() {
   setState(() {
     if (_counter > 0) {
       _counter--;
     }
```

© Editions ENI - All rights reserved

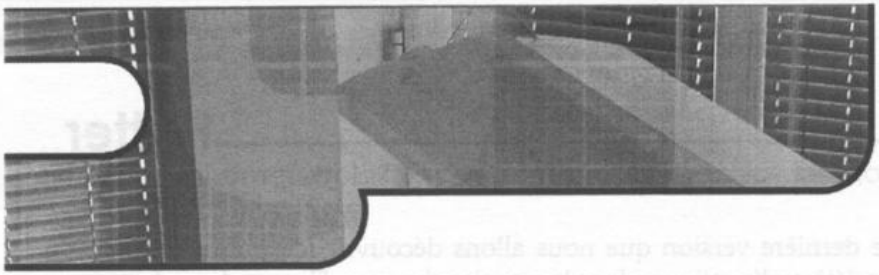
```
      if (_counter == 0) {
        _phrase = "BOUM";
        _tS = TextStyle(
          color: Colors.red,
          fontSize: 30,
          fontWeight: FontWeight.bold,
        );
        _couleur = Colors.amber;
      }
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              '$_phrase',
              style: _tS,
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ),
          ],
        ),
      ),
      backgroundColor: _couleur,
      floatingActionButton: FloatingActionButton(
        onPressed: _decrementCounter,
        tooltip: 'Compte à rebours',
        child: Icon(Icons.history),
      ),
    );
  }
}
```

Dans ce chapitre, au travers de la modification du code par défaut, il devient clair que la programmation d'applications multiplateformes est vraiment très facile et à la portée de tous grâce à Flutter.

Pour autant, certains lecteurs n'auront peut-être pas compris tout ce qui a été fait ici. Volontairement, quelques aspects ou mots de vocabulaire n'ont pas été développés afin de ne pas embrouiller les débutants. Il est donc temps que les choses deviennent moins obscures. Les deux prochains chapitres vont s'engouffrer dans la mécanique de Flutter avec l'étude de son langage, Dart.

© Editions ENI - All rights reserved



Chapitre 4
Dart, les fondamentaux

1. Introduction

Dart est un langage orienté objet moderne apparu en 2010 à l'initiative de deux développeurs de Google : Lars Bak et Kasper Lund.

À l'origine, ce projet avait comme objectif de remplacer JavaScript à moyen ou long terme, en raison des difficultés à faire évoluer ce dernier. Google a donc conçu Dart à partir de cette idée et la version 1.0 est sortie le 10 octobre 2011.

Le langage a même été adopté par ECMA International en 2014. Pour rappel, il s'agit d'une organisation européenne de standardisation en informatique. Parmi ses responsabilités, elle adopte des standards notamment pour les langages de programmation (y compris de scripts).

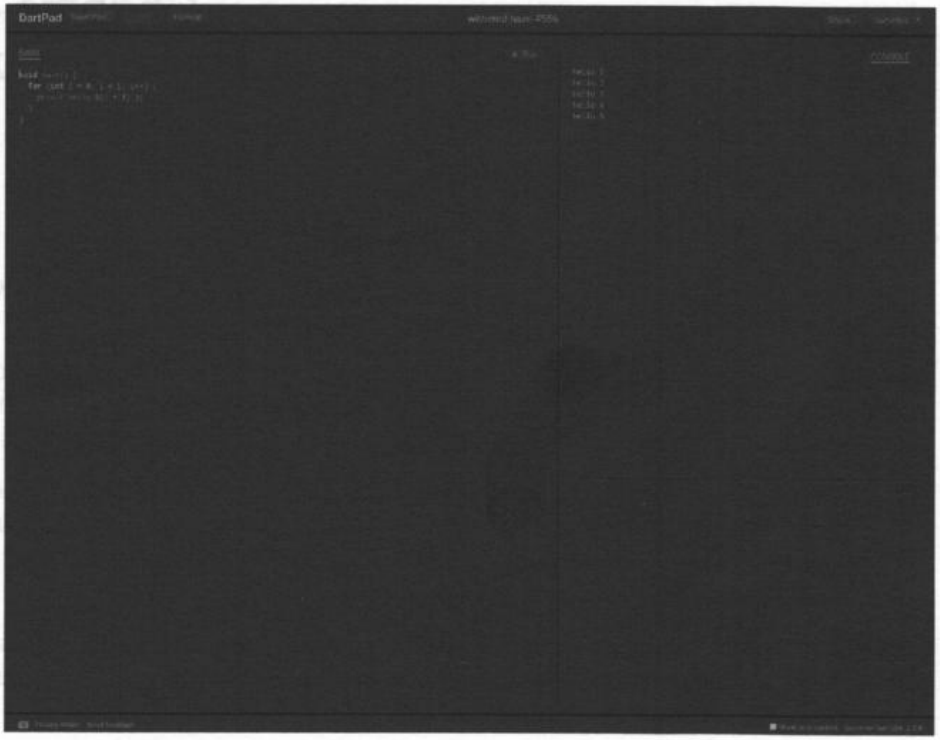
Malgré tous les efforts de Google, Dart n'a jamais percé et JavaScript reste bien établi sur son piédestal. Le langage est donc tombé petit à petit dans l'oubli jusqu'à l'arrivée de Flutter courant 2018.

La volonté de pousser le SDK au-devant de la scène, entre autres, a conduit Google à sortir une nouvelle version plus simple, plus rapide et plus intuitive de Dart en début d'année 2018, la 2.0.

C'est cette dernière version que nous allons découvrir ici, puisque, comme nous avons déjà pu l'expliquer dans le premier chapitre, Flutter s'appuie sur ce langage. Il est primordial d'en posséder les bases avant de pouvoir pleinement profiter de tous les avantages que nous procure Flutter.

Afin de mettre en œuvre et de tester les exemples qui vont suivre, il est bon de noter l'existence d'un outil extrêmement utile : `DartPad`.

Vous pourrez le trouver à l'adresse suivante : https://dartpad.dartlang.org/



DartPad de Google

© Editions ENI - All rights reserved

2. Syntaxe de base

Dans un premier temps, il convient de passer en revue, la syntaxe de base de Dart. Nous allons donc découvrir comment utiliser les variables, les constantes, les collections, les alternatives ainsi que les boucles. Le but du jeu n'est pas d'être exhaustif, mais bien de nous outiller avec les éléments indispensables pour la suite.

2.1 Variables

Dart supporte un certain nombre de types comme les nombres, les chaînes de caractères, les booléens, ou encore des collections comme les listes simples ou les listes fonctionnant avec un couple clé/valeur. Nous aborderons ces deux dernières notions un peu plus loin dans ce chapitre.

2.1.1 Les nombres

Syntaxe de base

Concernant les nombres, on peut noter la possibilité d'utiliser les entiers (Integer) ou les nombres réels (Double).

L'écriture va se faire de manière assez classique :

```
int monEntier;
double monReel;
```

Remarque

À ce moment-là, si la variable n'est pas initialisée avec une valeur choisie, elle prendra `null` comme valeur par défaut. Ce traitement est également valable pour les nombres (entier ou réel) puisqu'en Dart, tout est considéré comme objet. C'est un point notable puisque cela diverge de certains langages.

L'initialisation est réalisée comme suit :

```
int monEntier = 2;
double monReel = 1.856;
```

Héritage de la classe num

Les `int` et les `double` héritent de la classe `num` (les notions d'héritage seront développées plus amplement dans le prochain chapitre). Il est donc possible d'utiliser ce dernier pour garantir que la variable est bien un nombre de type `int` ou `double`. Si ce n'est pas le cas, une erreur de compilation se produira.

```
num unNombre = 318;
num unNombreDifferent = 3.265464;
```

La manipulation des nombres

Il est possible de mener différentes opérations sur les nombres telles que des conversions depuis une chaîne de caractères. Pour cela, il suffit d'utiliser la syntaxe suivante :

```
int.tryParse(valeurAConvertir) ;
```

Par exemple, pour convertir la chaîne de caractère « 32 » en entier et l'affecter immédiatement dans une variable il suffit d'écrire ceci :

```
int nombreDepuisChaineDeCaracteres = int.tryParse("32");
```

2.1.2 Les chaînes de caractères

Syntaxe de base

La déclaration et l'initialisation des chaînes de caractères est semblable à celle des nombres :

```
String maChaine = 'Hello World ! ';
```

La valeur de la chaîne est entourée de simple ou double *quote*.

La manipulation des chaînes de caractères

La conversion

Comme dans tant d'autres langages, il existe beaucoup de possibilités qui sont offertes concernant la manipulation de ces chaînes.

On peut noter d'abord la conversion depuis un nombre grâce à la fonction `toString()`. Sa syntaxe est des plus simple :

```
maChaine.toString() ;
```

© Editions ENI - All rights reserved

Il ne faut pas hésiter à utiliser cette fonction très puissante. D'autant qu'il est possible, comme décrit dans le chapitre suivant, de venir modifier le résultat de cette dernière. Voici un exemple :

```
int conv = 12;
String numConv = conv.toString();
```

La mise en majuscule ou minuscule

Des opérations classiques de mise en minuscules ou majuscules existent également, elles utilisent respectivement les syntaxes suivantes :

```
maChaine.toLowerCase()
```

ou

```
maChaine.toUpperCase()
```

En prenant l'exemple d'une chaîne de caractères nommée `minMaj` qui aurait pour valeur « Bonjour », il serait possible d'utiliser les deux fonctions comme ceci :

```
String minMaj = "Bonjour";
print(minMaj.toLowerCase());
print(minMaj.toUpperCase());
```

Remarque

La fonction `print()` sert à afficher la valeur d'un objet dans la console.

Le résultat de ces opérations sera le suivant :

Résultats des fonctions `toLowerCase()` et `toUpperCase()`

Le découpage de la chaîne de caractères

Enfin, il est également possible de découper une chaîne grâce à la fonction `substring()`. La syntaxe à utiliser est la suivante :

```
maChaîne.substring(valeurDeDepartIncluse, ValeurDeFinExcluse) ;
```

Concrètement, il est possible d'avoir l'exemple suivant :

```
print(minMaj.substring(0,3));   //Bon
```

Le premier paramètre est le point de départ de la scission. Dans l'exemple, le 'B' correspond à la position 0 dans la chaîne de caractères. Ce paramètre est donc inclus dans le résultat.

Le second paramètre, 3, est quant à lui le terminus et sera exclu. Dans l'exemple, il s'agit du 'j'.

La console affichera donc 'Bon'.

2.1.3 Les booléens

Un autre type très courant et qui sera extrêmement utile est le type booléen. Prenant les valeurs vrai ou faux, il servira notamment dans l'établissement d'alternatives.

L'écriture se fait comme suit :

```
bool choix1 = true;
bool choix2 = false;
```

2.1.4 Types dynamiques

Le type var

Enfin, il peut être intéressant d'évoquer les types `var` et `dynamic`. Ils sont utilisés quand on ne souhaite pas typer une variable au moment de sa déclaration. Au moment de l'initialisation, le système, en fonction de la valeur passée, déterminera lui-même le type approprié.

© Editions ENI - All rights reserved

Dans le cas de var, une fois que le système a déterminé le type, il ne sera plus possible d'en changer. Par exemple, après avoir réalisé l'initialisation suivante :

```
var uneVariable = "Hello";
```

Il sera impossible de tenter l'exécution du code ci-dessous sous peine de déclencher une erreur de compilation :

```
uneVariable = 1;
```

En effet, `uneVariable` est désormais connu par le système comme étant une chaîne de caractères, il est donc impossible de lui transmettre un entier.

Le type Dynamic

En revanche, les objets typés avec `dynamic` pourront voir leur type évoluer en fonction de celui des valeurs qui seront affectées.

```
dynamic val = 'Hello';
print(val);                    //Hello
print(val.runtimeType);        //String
val = 8;
print(val);                    //8
print(val.runtimeType);        //int
```

Dans l'exemple ci-dessus, `val` est reconnu dans un premier temps comme étant une chaîne de caractères puisque la valeur 'Hello' lui est affectée.

La troisième ligne le confirme grâce à la fonction `runtimeType` qui renvoie le type de la variable. Ici, c'est `String` qui ressort.

Tandis qu'après avoir affecté la valeur 8, le type est bien `int`.

2.2 Constantes

Dans le langage Dart, il existe deux mots-clés qui vont permettre de manipuler des données qui doivent rester constantes.

Le premier est `final`. Il rend une valeur affectée immuable. Elle ne pourra plus être changée. Par exemple :

```
final int constante1 = 12;
//constante1 = 13;
```

Après avoir initialisé cet entier avec la valeur 12, il sera impossible d'essayer de la passer à 13 sous peine de déclencher une erreur à la compilation.

Toutefois, le mot-clé `final` ne change pas les propriétés intrinsèques d'un objet. Par exemple, sur une collection de type liste qui sera abordée un peu plus tard dans ce chapitre, il est possible de rajouter des éléments au fur et à mesure grâce à une fonction appelée `add`.

```
final List<int> constante2 = [12, 15];
constante2.add(18);
```

Pour être encore plus restrictif, Dart met à disposition le mot-clé `const`. Ce dernier bloque également les opérations potentielles des objets et les fige. Il sera donc impossible, sous peine de déclencher une exception à la compilation, de reprendre la tentative effectuée ci-dessus, cette fois-ci avec `const` à la place de `final`.

```
const List<int> constante3 = [12, 15];
constante3.add(18);
```

Le résultat est le suivant dans la console :

```
Uncaught exception:
Unsupported operation: add
```

Exception levée

© Editions ENI - All rights reserved

2.3 Collections

Comme dans chaque langage, il sera commode de pouvoir manipuler des collections d'objets. Ce sont principalement les listes et les *maps* qui seront le plus utilisées.

2.3.1 Listes

Les listes permettent de définir une collection ou chaque item est associé à un index. Ce dernier démarre à partir de 0. Une liste sera de taille fixe ou modulaire.

Pour obtenir une taille déterminée, la déclaration de la liste s'écrit de la manière suivante :

```
List<int> listeDeNombres = new List(5);
```

Attention, il ne sera plus possible d'utiliser la fonction add, si bien connue. En revanche, il sera toujours possible de remplir cette liste de cette façon :

```
listeDeNombres[0] = 54;
listeDeNombres[1] = 62;
```

Dans cette configuration, cette liste pourrait être apparentée à un tableau.

Pour obtenir plus de flexibilité, il est possible de ne pas indiquer de taille prédéfinie.

```
List<int> listeDeNombres2= List<int>();
```

Il est intéressant de noter l'absence du mot-clé new qui n'est pas obligatoire dans la dernière version de Dart.

Dès lors, l'alimentation de la liste peut s'écrire comme suit :

```
listeDeNombres2.add(48);
listeDeNombres2.add(3);
listeDeNombres2.add(1);
listeDeNombres2.add(5);
listeDeNombres2.add(7);
```

Enfin, à l'instar de langages comme Java, il existe un grand nombre de fonctions inhérentes aux listes. Sans vouloir être exhaustif, en voici quelques-unes :

- `add(T valeur)` : comme vu précédemment, cette méthode permet d'ajouter un élément en fin de liste. Elle attend donc en paramètre un objet correspondant au type de la liste.

```
listeDeNombre2.add(12);
listeDeNombre2.add(38);
```

- `clear()` : cette méthode très utile permet de nettoyer et de vider une liste entièrement.

```
listeDeNombre2.clear();
```

- `indexOf(T item)` : cette fonction offre la possibilité de retrouver le premier item correspondant dans la liste. Pour cela, elle exige en paramètre l'objet recherché. Si celui-ci est présent, elle renvoie sa position via un `int`.

```
listeDeNombre2.indexOf(1);
```

- `remove(Object valeur)` : cette méthode permet d'effacer un élément de la liste. Afin d'y parvenir, il faudra passer en paramètre l'objet concerné. En cas de redondance, il n'y aura que la première occurrence qui sera supprimée.

```
listeDeNombre2.remove(12);
```

- `removeAt(int index)` : très semblable à la méthode précédente, à la différence que le paramètre attendu est un entier, qui correspond à la position de l'élément ciblé dans la liste.

```
listeDeNombre2.removeAt(5);
```

- `length` : renvoie la longueur de la liste.

```
listeDeNombre2.length;
```

- `sort()` : permet de trier la liste.

```
listeDeNombre2.sort(); //[1, 3, 5, 7, 48]
```

© Editions ENI - All rights reserved

2.3.2 Maps

Les *maps* sont un autre type de collection fonctionnant sur un système de couple clé/valeur. Traditionnellement, il y a deux façons de procéder pour créer une `Map`.

Premièrement, en passant par le constructeur :

```
mapDeValeurs['Cle1'] = 'Valeur1';

var secondeMapDeValeurs = Map();
```

Puis l'initialisation pourra s'écrire comme suit :

```
mapDeValeurs['Cle1'] = 'Valeur1';
mapDeValeurs['Cle2'] = 'Valeur2';

secondeMapDeValeurs[1] = 'ValeurDifferente1';
secondeMapDeValeurs[2] = 'ValeurDifferente2';
```

Il est également possible d'initialiser directement notre collection :

```
var mapInitilisee = {'cle1':'valeur1','cle2':'valeur2'};
```

En termes de méthodes et fonctions associées, certaines sont communes aux listes. On retrouve ainsi `clear()`, `remove(Object clé)` ou `length()`.

Hormis celles-ci, il existe quelques méthodes importantes à connaître :

- `values()` : retourne toutes les valeurs de la `map` sans les clés.
- `putIfAbsent(C Cle, X isAbsent())` : permet de vérifier si la clé passée est déjà présente dans la `map` et si ce n'est pas le cas, réalise une insertion de l'élément fourni.

En poursuivant avec `mapInitialisee`, cela donnerait :

```
mapInitilisee.putIfAbsent('cle3', () => 'valeur3');
```

- `update(C cle, N update(V valeur))` : permet de mettre à jour une valeur en la recherchant par sa clé :

```
mapInitilisee.update('cle3',(String) => 'valeurDifferente3');
```

Il est même possible de combiner `update` et `ifAbsent` :

```
mapInitilisee.update('cle4',(String) => 'ValeurSiPresent',
ifAbsent: () => 'ValeurSiAbsent');
```

2.4 Alternatives

Dans le classement des besoins couramment rencontrés, les alternatives tiennent le haut du pavé. En effet, dans un programme où la logique est essentielle, pouvoir choisir entre deux ou plusieurs solutions est indispensable.

Il existe plusieurs façons de gérer les conditions. La première est sous la forme d'un « si, sinon » que l'on appellera `if`, ci-dessous. La seconde est constituée d'un « au cas où », le `switch`.

2.4.1 If

L'utilisation d'`if` est très courante. Elle correspond littéralement au si, sinon si, sinon en français. Le si et le sinon si, attendent une condition, qui si elle se révèle exacte, déclenchera l'exécution d'un bout de code. Le sinon, lui, ne vient que clôturer le débat et n'exécutera du code que si aucune des conditions qui précèdent n'a pu être vérifiée.

```
if (footballeur) {
  print("Je pratique le football.");
} else if (nageur) {
  print("Je pratique la natation.");
} else {
  print("Je ne fais aucun sport !");
}
```

Dans cet exemple, deux booléens sont créés : `footballeur` et `nageur`. En fonction de leur valeur, la première condition sera vérifiée si `footballeur` possède la valeur `true`. La seconde, si c'est `nageur` qui est `true`, mais pour cela `footballeur` devra lui être `false`. La dernière condition sera appliquée si `footballeur` et `nageur` ont `false` pour valeur.

© Editions ENI - All rights reserved

2.4.2 Switch

La seconde façon de réaliser une alternative et d'utiliser `switch`. En traduction brute, il s'agirait du « au cas où ». Le principe est sensiblement le même que pour `if`, à la différence que tous les cas vont être exposés sans ordre de priorité.

```
switch (sport) {
  case 'football':
    print("Je pratique le football.");
    break;
  case 'natation':
    print("Je pratique la natation.");
    break;
  default:
    print("Je ne fais aucun sport !");
}
```

`Switch` surveille une variable de type `String` appelée `sport`. Dans le cas où `sport` est égal à la valeur `football`, c'est le premier cas qui est exécuté. Autrement, la seconde valeur possible est `natation`. Si jamais une autre valeur est attribuée à `sport` alors c'est le cas par défaut qui prendra le pas.

2.4.3 Opérateur ternaire

L'opérateur ternaire est une autre forme d'alternative qui est, en réalité, une forme simplifiée d'`if`. Elle s'écrit avec une syntaxe particulière où l'on retrouve d'abord la condition suivie d'un point d'interrogation. Derrière celui-ci, on pourra inscrire le code à exécuter dans le cas où la condition serait vérifiée. Puis, il faudra placer la ponctuation deux points et terminer par le code qui s'exécutera si la condition n'est pas remplie.

```
footballeur ? print("Je pratique le football.") :
print("Je ne fais pas de football !");
```

2.4.4 Opérateur de nullité

L'opérateur de nullité est très proche de l'opérateur ternaire. La différence majeure réside dans l'utilisation du double point d'interrogation. Son unique intérêt est de vérifier la nullité d'un objet. C'est également une version simplifiée d'`if`.

Voici une méthode qui affiche une chaîne de caractères en fonction du paramètre reçu :

```
void pratiqueDuFootball(bool foot) {
  if (foot == null) {
    print("Je ne pratique pas le football.");
  }
}
```

Dans cette construction classique, c'est la nullité qui est vérifiée. Grâce à l'opérateur de nullité, il est possible d'utiliser la syntaxe suivante :

```
void pratiqueDuFootball(bool foot) =>
foot ?? print("je ne pratique pas le football !");
```

Remarque

Dans ce cas, la syntaxe de la méthode diffère quelque peu. Les accolades ont disparu au profit de => qui vient raccourcir et factoriser cette expression.

2.5 Boucles

Hormis les alternatives, les boucles sont essentielles dans le cadre de la programmation d'applications. Les boucles permettent l'itération sur une collection par exemple. Dart propose différents types de boucles comme `for`, `foreach`, `while` ou encore `do while`.

2.5.1 For

La boucle for permet d'itérer de manière précise en gardant le contrôle. Pour cela, un paramètre de type `int` sert de compteur, borné à une limite définie et pour lequel une cadence, un pas est précisé.

```
for (int i = 0; i < 5; i++) {
  print("i vaut $i");
}
```

© Editions ENI - All rights reserved

Le résultat obtenu sera :

```
i vaut 0
i vaut 1
i vaut 2
i vaut 3
i vaut 4
```

Résultat d'une boucle for

2.5.2 Foreach

`Foreach` est d'une écriture plus simple. Plus facile à utiliser donc, mais plus limité également. Il suffit de créer une variable et d'intégrer le mot-clé `in` suivi de la collection sur laquelle l'itération va se dérouler.

```
var maCollection = ["valeur1", "valeur2", "valeur3"];

for (var i in maCollection) {
   print(i); // valeur1 valeur2 valeur3
}
```

Ici, il n'est pas possible de définir nous-mêmes la limite. Elle correspond, de facto, à la taille de la collection.

2.5.3 While

`While` est utilisé pour obtenir une répétition qu'il serait possible de traduire par tant que.

Ce type de boucle est notamment utilisé avec des booléens. Il continue son exécution jusqu'à ce que la valeur du booléen change. À noter qu'il ne se déclenche qu'après la vérification de la condition.

```
bool stop = false;
var x = 0;

while (stop) {
```

```
  if (x == 5) {
    stop;
  }
  x++;
}
```

Préalablement, deux variables sont créées. La première est un booléen appelé stop et initialisé avec `false`. La seconde nommée x est une variable qui vaut 0.

La boucle va s'exécuter tant que x n'aura pas atteint 5. À ce stade, le booléen passera à `true` et la boucle prendra fin.

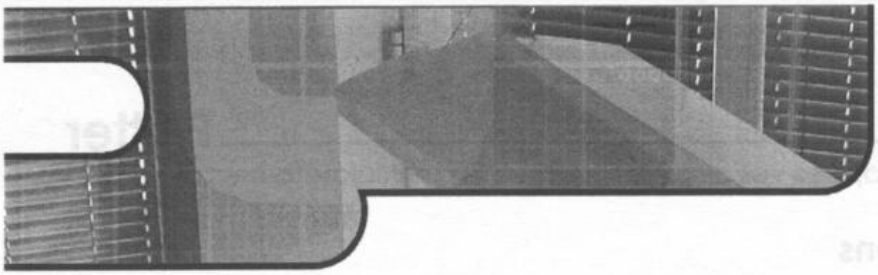
2.5.4 Do While

Sa structure est quasi-identique à celle de `while` à la différence près que la condition est vérifiée après et non dès le début. Ce qui signifie que, quoi qu'il arrive, le code présent sera exécuté au moins une fois.

```
bool stop = false;
var x = 0;

do {
 if (x == 5) {
   stop;
 }
 x++;
}while (stop);
```

© Editions ENI - All rights reserved



Chapitre 5
Dart, notions avancées

1. Introduction

Après avoir exposé les fonctionnalités de base du langage Dart, il est utile à présent d'en découvrir encore davantage pour être parfaitement à l'aise avec le sujet.

Les concepts et notions détaillés dans ce chapitre sont suffisants dans le cadre de cet ouvrage afin de découvrir Flutter et pour être suffisamment à l'aise. Malgré tout, il restera encore beaucoup de choses à découvrir concernant Dart. Il n'est pas envisageable de décrire l'ensemble des possibilités de ce langage sur deux chapitres. Il faut comprendre qu'il s'agit d'un langage à part entière et qu'un ouvrage entier pourrait lui être consacré.

Toutefois, rien n'empêche de découvrir les bases (déjà conséquentes) au travers de nombreux thèmes. Le premier abordé sera les fonctions. Le mot est volontairement un peu généraliste. Par la suite, un aspect volumineux du langage sera vu à travers la programmation orientée objet (POO). Ici, les classes et tout ce dont elles sont composées seront étudiées. Leurs interactions (associations, héritages...) permettront de mieux comprendre l'utilité de cette organisation. C'est aussi le moment idéal pour comprendre comment gérer les erreurs qui peuvent potentiellement survenir pendant l'exécution d'un programme (Exceptions). Enfin, un coup d'œil rapide sera donné sur la notion de traitement asynchrone.

2. Fonctions

2.1 Introduction

Il est temps maintenant d'aborder une première notion indispensable : il s'agit des fonctions. Elles vont permettre un découpage précis du code et le regroupement de fonctionnalités. Il existe plusieurs types de fonctions partant de la fonction simple, passant par la fonction avec paramètre ou encore la fonction avec retour et même la fonction anonyme.

2.2 Fonction simple

Une fonction, dans sa forme la plus simple, n'attend pas de paramètre et ne renvoie rien. Cette fonction est de type `void`. Dans le jargon informatique, elle se nomme procédure. Dans le code présenté lors du chapitre précédent, elle a été rencontrée au travers notamment d'un élément crucial, `main`, le point d'entrée du programme. Pour rappel, sa syntaxe est la suivante :

```
void main() {
 //Point d'entrée du programme
}
```

Voici un autre exemple. Si le but du programme est d'afficher le traditionnel « Hello World ! », il est possible d'envisager de conserver `main` et d'écrire une fonction qui sera appelée et dont le rôle sera d'afficher cette phrase. Voilà ce que cela peut donner :

```
void main() {
 //Point d'entrée du programme
 afficher();
}

void afficher() {
 print("Hello World !");
}
```

© Editions ENI - All rights reserved

Le programme ci-dessus s'exécute, car il contient une procédure `main`. La première et unique action effectuée est d'appeler une fonction se nommant `afficher()`. Celle-ci est écrite juste en dessous. Elle est de type `void` puisqu'elle ne renvoie rien. Son seul but est d'afficher dans la console le contenu de la fonction `print()`. Le code est bien scindé et commence à avoir une structure plus ordonnée.

2.3 Fonctions avec retour

Une fonction avec retour se comporte globalement de la même manière que la fonction simple hormis qu'une réponse va être renvoyée.

Ces fonctions sont utilisées dans un grand nombre de situations comme pour la réalisation d'un calcul, la vérification d'une condition, etc.

Les deux différences notables interviennent dans la signature puisque le mot `void` disparaît au profit du type qui sera retourné, ainsi qu'à la fin de la fonction où le mot `return` fait son apparition. Ce dernier correspond au retour c'est-à-dire l'information qui sera renvoyée à la fin du traitement. Pour continuer avec l'exemple précédent, une fonction intitulée `obtenirUnPrenom()` est créée. Son but est de renvoyer un prénom sous la forme d'une chaîne de caractères (`String`) :

```
void main() {
 //Point d'entrée du programme
 afficher();
 String prenomRecupere = obtenirUnPrenom();
 print(prenomRecupere);
}

void afficher() {
 print("Hello World !");
}

String obtenirUnPrenom() {
 String prenom = "Julien";
 return prenom;
}
```

L'exécution de la fonction permet de retourner le prénom « *Julien* » qui est par la suite réaffecté dans une variable nommée `prenomRecupere`. Le type de retour est bien précisé en début de fonction, il s'agit de `String`. Ce qui rend la chose d'autant plus intéressante, c'est que cela fonctionne avec n'importe quel type, il n'y a pas de limite.

2.4 Fonctions avec paramètres

2.4.1 Forme classique

Les fonctions ont parfois besoin de recevoir des paramètres afin de les exploiter dans le cadre de leur traitement. Les paramètres peuvent varier en type comme en nombre. Ils s'écrivent à l'intérieur des parenthèses de la signature. Pour reprendre l'exemple montré ci-dessus, l'idée va être de le faire évoluer pour appeler une fonction qui attendra qu'on lui transmette un prénom. Elle le mettra ensuite en majuscule avant de le renvoyer. Ainsi, plutôt que d'avoir « *Hello World !* », le prénom fraîchement formaté viendra remplacer le mot « *World* ».

```
void main() {
 //Point d'entrée du programme
 print(mettrePrenomEnMajuscule("Julien"));
}

String mettrePrenomEnMajuscule(String prenom) {
 String retour = "Hello ${prenom.toUpperCase()} !";
 return retour;
}
```

Dans le code ci-dessus, la fonction `mettreEnMajuscule(String prenom)` est créée et attend en paramètre une chaîne de caractères (`String`). Après l'avoir transformé en majuscule, elle le renverra en le rajoutant à « Hello ». Ici, `main` appelle la fonction au travers de `print()` et transmet le prénom « *Julien* ». Finalement, en console le résultat obtenu sera le suivant :

Hello JULIEN !

© Editions ENI - All rights reserved

Remarque

*Il est intéressant de noter la syntaxe de la première ligne qui se trouve à l'intérieur de la fonction `mettrePrenomEnMajuscule()`. Une chaîne de caractères (`retour`) va correspondre à la concaténation, c'est-à-dire l'addition, du mot « Hello » et du prénom mis en majuscule. Pour réaliser cette étape en une seule ligne, une **interpolation** est utilisée. Elle est toujours précédée du symbole « `$` » quand il s'agit de rappeler une simple variable. Pour un appel de fonction, il faudra rajouter les accolades comme dans l'exemple.*

2.4.2 Paramètres optionnels

Il est aussi possible de rendre certains paramètres optionnels. Cela procure une grande souplesse aux fonctions qui peuvent réaliser un traitement différencié en fonction des besoins. Par exemple :

```
void fonctionAvecParametreOptionnel(String nom, String prenom, [int age]) {
 if(age != null) {
   print("nom : $nom, prénom : $prenom, âge : $age");
 } else {
   print("nom : $nom, prénom : $prenom");
 }
}
```

Cette fonction prend trois paramètres. Deux sont obligatoires, le nom et le prénom. Le dernier est optionnel, il s'agit de l'âge qui se retrouve alors entouré de crochets. Au moment de l'appel de `fonctionAvecParametreOptionnel()`, l'IDE offre donc les deux possibilités :

```
String nom = "TRILLARD";
String prenom = "Julien";
int age = 38;
//Si on ne connait pas l'âge
fonctionAvecParametreOptionnel(nom, prenom);
//Si on le connait
fonctionAvecParametreOptionnel(nom, prenom, age);
```

La contrepartie est de prévoir ce que devient ce paramètre en fonction de sa réception ou non. Dans le cas présenté ci-dessus, une condition vérifie sa nullité. La chaîne de caractères produite variera en conséquence. Finalement, en console, le résultat sera l'un ou l'autre :

nom : TRILLARD, prénom : Julien
nom : TRILLARD, prénom : Julien, âge : 38

2.5 Fonctions anonymes

Les fonctions qui ont été décrites précédemment dans ce chapitre ont un point en commun, elles portent toutes un nom. À l'inverse, les fonctions anonymes n'en ont pas. Elles sont donc inscrites dans le code pour un besoin unique puisqu'elles ne pourront pas être appelées à un autre endroit. L'exemple suivant manipule une liste de prénoms qu'il faut parcourir. Pour cela, une fonction `forEach()` est utilisée, à l'intérieur de laquelle une fonction anonyme est mise en place :

```
var prenoms = ["Julien", "Jules", "Hugo"];

prenoms.forEach((item) {
 print("${prenoms.indexOf(item)}: $item");
});
```

Dans la console, le rendu est ceci :

0: Julien
1: Jules
2: Hugo

© Editions ENI - All rights reserved

2.6 Fonctions fléchées ou fonctions lambda

Les fonctions fléchées (ou fonctions lambda) sont exactement identiques aux fonctions classiques à la différence que la syntaxe est plus courte. Pour les mettre en place, il suffit de remplacer les accolades par une flèche composée du signe égal et d'un chevron (=>).

Pour reprendre les exemples présentés plus haut, il aurait été possible d'écrire une fonction fléchée pour `afficher()` :

```
//La fonction afficher()
void afficher() {
 print("Hello World !");
}
//peut s'écrire aussi :
void afficher2() => print("Hello World !");
```

Ou pour la fonction `mettrePrenomEnMajuscule()` :

```
//Avant
String mettrePrenomEnMajuscule(String prenom) {
 String _retour = "Hello ${prenom.toUpperCase()} !";
 return _retour;
}
//Equivalent en fonction fléchée :
String mettrePrenomEnMajuscule2(String prenom) => "Hello
${prenom.toUpperCase()} !";
```

Sur le dernier exemple, il faut noter un point important. Il s'agit de la disparition du mot `return`. La flèche a donc la particularité de simplifier l'écriture des fonctions sans en perturber la compréhension. L'inconvénient est que ce type de syntaxe n'est pas adaptée aux fonctions complexes qui nécessitent de longs traitements.

3. Programmation orientée objet (POO)

3.1 Introduction

La programmation orientée objet, couramment appelée POO, est un paradigme de programmation. L'idée principale est de faire en sorte que la représentation que l'être humain a du monde qui l'entoure puisse être traduite en code informatique. Pour cela, il convient de représenter une problématique au travers des acteurs, éléments ou objets en jeu.

Si le but d'un logiciel est de gérer des clients et des fournisseurs, le plus simple va être d'obtenir une représentation de chacun d'eux sous forme de code exploitable. Pour commencer, il faudra créer des classes qui seront en quelque sorte le moule de chaque élément. À l'intérieur, des caractéristiques intrinsèques à chaque objet seront décrites, il s'agit des attributs. Ces objets disposeront également de fonctionnalités (méthodes). Par la suite, elles seront capables de travailler ensemble soit par le biais d'un héritage ou d'une association.

3.2 Classe

La classe peut être considérée comme un moule qui va servir à créer des objets appelés instances. Pour créer une classe, il suffit d'ouvrir un nouveau fichier Dart. Une fois ouvert, il suffit d'écrire le mot-clé `class` puis de nommer celle-ci et enfin d'ouvrir des accolades :

```
class Personne {

}
```

© Editions ENI - All rights reserved

Une classe peut être créée un peu n'importe où. À la suite d'une fonction `main` par exemple. Il s'agit alors d'une classe interne. Dans le cadre d'une bonne pratique, il est conseillé de créer des fichiers indépendants qui ne contiendront chacun qu'une classe. Pour les utiliser dans un autre fichier, celui contenant la procédure `main` par exemple, il faudra alors réaliser un `import` :

```
import 'Personne.dart';

void main() {
}
```

3.3 Attributs

Dans une classe, les attributs pourraient être considérés comme les caractéristiques de l'objet. Une personne possède un nom, un prénom, une date de naissance... Une voiture, elle, a une couleur, une marque, un modèle...

À l'intérieur de la classe, l'enjeu va être de déclarer ces attributs de la même façon que des variables quelconques. Ils possèderont un type et seront explicitement nommés. Toutefois, il existe une particularité à respecter, l'encapsulation.

Il s'agit de protéger ces éléments du contact extérieur afin de garantir que ce qu'ils contiennent ne subira pas de modifications sauvages.

Pour garantir cette intégrité, les attributs doivent être déclarés privés ; c'est-à-dire qu'ils ne seront exploitables que dans la classe où ils se trouvent. Dans beaucoup de langages, il existe des mots-clés pour cela comme `private` en Java.

En Dart, il n'y en a pas. En revanche, il est possible de mettre en préfixe du nom de l'attribut un *underscore* (tiret bas) qui produira le même effet.

Voici la classe créée précédemment à laquelle les attributs **nom** et **prénom** ont été ajoutés :

```
class Personne {
 String nom;
 String _prenom;
}
```

Ces deux attributs sont des chaînes de caractères. Ils s'appellent `nom` et `_prenom` et possèdent respectivement une portée publique et privée (due à la présence de l'*underscore*).

Théoriquement, les deux attributs auraient dû être privés. Mais dans la suite de l'exemple, il sera utile de constater la différence quand une personne sera créée dans un autre fichier.

Justement, comment créer une personne ? Il existe un outil autorisant cela et son nom est des plus explicite : le constructeur.

3.4 Constructeur

Le constructeur est le moyen le plus aisé de créer une nouvelle instance d'une classe. Il s'écrit en reprenant le nom de la classe suivi, entre parenthèses, des différents paramètres qui viendront alimenter les attributs.

Dans la classe `Personne`, le constructeur le plus complet pourrait être :

```
Personne(String nom, String prenom) {
 this.nom = nom;
 this._prenom = prenom;
}
```

Le mot-clé `this` vient préciser qu'il s'agit bien d'une variable courante de la classe (un attribut). Sur la ligne `this.nom = nom`, la partie `this.nom` correspond à l'attribut auquel est affectée la valeur `nom` reçue en paramètre. Pour la ligne suivante, `this` n'était pas nécessaire puisque le nom de l'attribut et celui du paramètre sont différents, il n'y a donc pas de confusion possible.

Cette syntaxe de constructeur est assez lourde. Dart permet de réduire le code pour obtenir une formulation équivalente plus courte :

```
Personne(this.nom, this._prenom);
```

Parfois, il est utile de posséder plusieurs constructeurs qui vont prévoir les différents cas possibles. Dans l'exemple d'une personne, peut-être que dans un premier temps, seul le prénom sera connu puis plus tard le nom. Au moment où il va falloir créer une nouvelle personne, il ne sera pas possible d'utiliser le constructeur décrit juste au-dessus, il est trop complet et impose de connaître aussi le nom.

© Editions ENI - All rights reserved

Dans beaucoup de langages, comme Java, la démarche consistera à écrire autant de constructeurs que nécessaire. Il est facile d'imaginer que sur des classes importantes, qui comportent un grand nombre d'attributs, cela représente une charge d'écriture colossale. Et cela, sans compter sur les risques d'erreurs ou d'oublis.

En Dart, il existe une forme de constructeur plus souple qui évite cette fastidieuse entreprise. En utilisant les accolades autour des paramètres optionnels, il est possible de n'avoir qu'un seul constructeur qui remplisse ce rôle au lieu de plusieurs. Dans l'exemple, où le nom n'est pas toujours connu d'avance, cela pourrait donner un constructeur comme celui-ci :

```
Personne(this._prenom, {this.nom});
```

Remarque

Il faut noter un point important tout de même. Les éléments optionnels doivent être placés après ceux qui sont obligatoires. Ici, le prénom est attendu quoi qu'il arrive. Il est donc placé avant le nom qui, lui, est facultatif pour l'instant.

Dans tous les cas, si rien n'est spécifié dans la classe, c'est-à-dire qu'aucun constructeur n'a été écrit, il y aura un constructeur par défaut qui correspond à ceci :

```
Personne() {
}
```

Ce constructeur vide suppose qu'aucun paramètre n'est connu. En d'autres termes, qu'au moment où une nouvelle personne va être créée, rien n'est su à son sujet. Si le développeur ne précise pas de constructeur dans la classe, celui-ci va exister par défaut (même s'il n'est pas explicitement écrit).

L'inconvénient de n'en mettre aucun ou d'ajouter uniquement le constructeur par défaut, c'est qu'il sera impossible, dans un premier temps, de créer une personne en renseignant son nom et son prénom. La construction se fera obligatoirement sans paramètre.

Pour parer à cela, il est possible de spécifier un constructeur dont tous les paramètres sont optionnels.

Pour finir sur ce sujet, il est également possible de créer des constructeurs nommés. En principe, en Dart, il n'y a besoin que d'un seul constructeur. Cela étant dû aux possibilités offertes avec les paramètres optionnels. Cependant, il pourrait être pratique d'en posséder plusieurs et de leur donner un nom afin de rendre leur utilité très claire.

Dans l'exemple de la classe Personne, si un troisième attribut était ajouté comme le numéro de téléphone ou bien l'adresse, il pourrait être utile d'avoir un constructeur complet, mais aussi un autre pour récupérer le nom et le prénom :

```
class Personne {
 String nom;
 String _prenom;
 String telephone;
 String adresse;

 //Un constructeur pour tous les attributs avec le prénom obligatoire
 Personne(this._prenom, {this.nom, this.telephone, this.adresse});

 //Un constructeur nommé etatCivil qui n'attend que le prénom et le nom
 Personne.etatCivil(this._prenom, this.nom);
}
```

Ci-dessus, il existe deux constructeurs. Le premier regroupe l'ensemble des attributs où seul le prénom est obligatoire, les autres sont optionnels. Le second est un constructeur nommé grâce à une syntaxe du type `Personne.xxx`. Il porte le nom `etatCivil` afin de rendre plus explicite son rôle puisqu'il n'attend que le prénom et le nom.

3.5 Instanciation

Maintenant que les constructeurs sont mis en place, il est temps de créer une nouvelle instance de la classe `Personne`. Pour cela dans la procédure `main` du programme, il faut écrire :

```
import 'Personne.dart';

void main() {

 String nom = "TRILLARD";
```

© Editions ENI - All rights reserved

```
  String prenom = "Julien";
  String telephone = "0203040506";
  String adresse = "ENI Edition, Nantes";

  //Appel du constructeur en utilisant différents paramètres
  Personne u1 = Personne(prenom);
  Personne u2 = Personne(prenom, nom: nom);
  Personne u3 = Personne(prenom, nom: nom, adresse: adresse,
telephone: telephone);
  //Utilisation du constructeur nommé
  Personne u4 = Personne.etatCivil(prenom, nom);
}
```

Dans l'exemple ci-dessus, quatre personnes sont créées. Grâce à l'importation réalisée précédemment, il est possible de créer des variables de type `Personne`. Comme pour les autres variables, il faut également leur donner un nom. Ici, il s'agit de `u1`, `u2`, `u3` et `u4`.

Maintenant que les variables sont posées, il faut procéder à l'instanciation pour chacune d'elle. Pour cela, de l'autre côté du signe « = », un appel au constructeur de la classe sera effectué.

Dans l'exemple, `u1` appelle le constructeur dans sa forme la plus simple. Seul le paramètre obligatoire est fourni (le prénom).

Pour `u2` et `u3`, il est fait appel au même constructeur, mais cette fois-ci en incluant, en plus, des paramètres optionnels, respectivement le nom pour `u2` puis le nom, l'adresse et le téléphone pour `u3`.

Il est important de constater une différence notable dans l'écriture des paramètres obligatoires et optionnels. Ces derniers, dans la mesure où ils sont dispensables, peuvent être placés dans n'importe quel ordre. Il convient alors, pour éviter toute confusion ou erreur, de faire précéder le paramètre par son nom. Il prend la forme `nomDuParamètreAttendu : valeurDuParamètre`. Ainsi, pour la personne `u3`, l'une ou l'autre forme suivante sont correctes :

```
Personne u3 = Personne(prenom, nom: nom, adresse: adresse,
telephone: telephone);

//ou
```

```
Personne u3 = Personne(prenom, telephone: telephone, nom: nom,
adresse: adresse);
```

Remarque

Dans le cas où les paramètres optionnels ne sont pas initialisés, ils prendront de facto la valeur `null`.

Enfin, la personne `u4` est instanciée grâce au constructeur nommé `Personne.etatCivil`. Le nom, qui était optionnel dans le constructeur classique, est devenu obligatoire. Il n'est d'ailleurs pas précédé de « nom : ». Toute l'utilité du constructeur nommé se retrouve ici. Permettre la manipulation d'autres constructeurs en leur donnant un nom explicite.

Remarque

Contrairement à d'autres langages, l'instanciation se passe de l'écriture du mot-clé « new » et cela depuis la version 2.0 de Dart. Malgré tout, pour les développeurs ayant l'habitude de l'utiliser, il est toujours possible de l'inclure. Pour `u1`, cela donnerait :

```
Personne u1 = new Personne(prenom);
```

3.6 Accesseurs et mutateurs (getters et setters)

Maintenant que quelques personnes ont été créées, il pourrait être intéressant de les manipuler. Comment réobtenir le prénom de `u1` ou le téléphone de `u3` ?

La réponse dépend avant tout de la visibilité de l'attribut en question. S'il est public, il sera possible d'y accéder directement :

```
print("Le numéro de téléphone de u3 est : ${u3.telephone}");
```

Dans la console, le résultat est :

Le numéro de téléphone de u3 est : 0203040506

© Editions ENI - All rights reserved

En revanche, s'il est privé, il sera impossible d'y avoir accès. Dans ce cas-là, il faudra passer par une méthode publique qui permettra de lire la valeur contenue dans l'attribut. Il s'agit d'un *getter*. Pour la modifier, il faudra une fonction dédiée, un *setter*.

Android Studio permet de les créer automatiquement et sans effort.

- Il suffit pour cela d'appuyer sur [Alt] + [Inser] ou en faisant un clic droit sur la page puis **Generate...**.
- Sur la fenêtre suivante, il faut cliquer sur **Getter and Setter**.

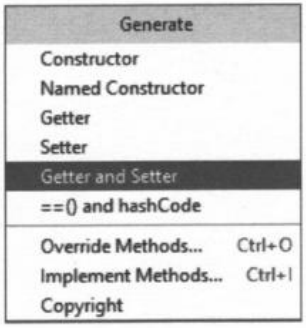


Figure 1 - Génération automatique de code – Android Studio

La boîte de dialogue qui va s'ouvrir permet de choisir les attributs sur lesquels des *getters* et *setters* doivent être créés. Dans la classe `Personne`, seul le prénom est concerné puisque c'est le seul attribut privé :

Figure 2 - Génération de getters et setters

© Editions ENI - All rights reserved

Après avoir choisi, il suffit de cliquer sur **OK**, le code suivant devrait apparaître dans la classe Personne :

```
//Getter
String get prenom => _prenom;

//Setter
set prenom(String value) {
 _prenom = value;
}
```

Le principe de l'encapsulation est respecté. L'attribut est protégé par sa visibilité privée et rien ne peut le modifier directement. Le fait d'y avoir accès de manière détournée en lecture (*getter*) ou en écriture (*setter*) respecte ce principe.

Dans `main`, il est dorénavant possible de voir le contenu du prénom de `u1` :

```
print("Le prénom de u1 est : ${u1.prenom}");
```

Le résultat affiché dans la console est :

Le prénom de u1 est : Julien

Il faut remarquer qu'ici ce n'est pas l'attribut qui est directement appelé, mais le *getter*. D'ailleurs, pour rappel, l'attribut est `_prenom` et non pas `prenom`. Il s'agit bien du *getter*.

Remarque

Il est inutile de mettre en place des getters/setters pour les attributs publics, car ils existent déjà de manière implicite.

3.7 Association

Parfois, il arrive que les objets travaillent ensemble. On parle dans ce cas d'association. L'idée ici n'est pas de traiter en profondeur du sujet, mais seulement de donner un minimum de bases à ceux et celles qui découvrent la programmation orientée objet.

Cette relation se veut logique. Dans l'exemple traité plus haut, la classe `Personne` possède un attribut `adresse`. À l'heure actuelle, il est de type `String`. Pourtant, il pourrait être avantageux de créer une classe `Adresse`.

Travailler avec une chaîne de caractères suppose de devoir écrire dans celle-ci l'ensemble des informations que comporte une adresse comme le numéro de voie, le nom de la voie, le code postal et le nom de la commune. Le risque est de se retrouver avec une quantité importante de texte mal formaté et peu accessible. S'il faut, à un moment, retrouver toutes les personnes d'une même commune, cela sera compliqué.

L'autre solution est de créer une classe `Adresse`. Elle comporterait les caractéristiques évoquées précédemment au travers de ses attributs. Reste une problématique : comment lier les deux classes par la suite ?

Avant de répondre à cette question, la classe `Adresse` doit être mise en place.

```
class Adresse {
 int _numeroDeVoie;
 String _nomDeLaVoie;
 String _codePostal;
 String _nomDeLaCommune;

 Adresse({int numeroDeVoie, String nomDeLaVoie, String codePostal,
     String nomDeLaCommune}) {
   this._numeroDeVoie = numeroDeVoie;
   this._nomDeLaVoie = nomDeLaVoie;
   this._codePostal = codePostal;
   this._nomDeLaCommune = nomDeLaCommune;
 }

 String get nomDeLaCommune => _nomDeLaCommune;

 set nomDeLaCommune(String value) {
   _nomDeLaCommune = value;
 }

 String get codePostal => _codePostal;

 set codePostal(String value) {
   _codePostal = value;
 }

 String get nomDeLaVoie => _nomDeLaVoie;
```

© Editions ENI - All rights reserved

```
  set nomDeLaVoie(String value) {
    _nomDeLaVoie = value;
  }

  int get numeroDeVoie => _numeroDeVoie;

  set numeroDeVoie(int value) {
    _numeroDeVoie = value;
  }

}
```

La classe comporte les quatre attributs nécessaires avec une visibilité privée, un constructeur dont tous les paramètres sont optionnels, ainsi que les *getters* et *setters*.

Il faut maintenant associer la classe `Adresse` à la classe `Personne`. Dans la section Classe de ce chapitre, la notion d'importation a été évoquée. L'importation permet d'utiliser une classe à l'intérieur d'un autre fichier y compris une autre classe.

Une autre question se pose de fait : est-ce un `import` d'`Adresse` dans `Personne` ou l'inverse qu'il faut mettre en œuvre. Les associations ont toujours un sens. C'est un élément fondamental. Dans le cas pratique présenté, la réponse est déjà donnée. Il y avait déjà un attribut « adresse » dans `Personne`. `L'import` doit donc se faire de ce côté. Grâce à celui-ci, l'attribut peut être modifié :

```
import 'Adresse.dart';

class Personne {
 String nom;
 String _prenom;
 String telephone;
 //String adresse;
 //L'adresse n'est plus du type String, mais fait référence à la classe Adresse
 Adresse adresse;


 //Un constructeur pour tous les attributs avec le prénom obligatoire
 Personne(this._prenom, {this.nom, this.telephone, this.adresse});

 //Un constructeur nommé etatCivil qui n'attend que le prénom et le nom
 Personne.etatCivil(this.prenom, this.nom);
```

```
  String get prenom => _prenom;

  set prenom(String value) {
    _prenom = value;
  }

}
```

Il est temps de voir comment instancier cela. La personne u3 possède une adresse. Son instanciation évolue sous cette forme :

```
import 'Adresse.dart';
import 'Personne.dart';

void main() {
  String nom = "TRILLARD";
  String prenom = "Julien";
  String telephone = "0203040506";
  //String adresse = "ENI Edition, Nantes";

  //Personne u3 = Personne(prenom, nom: nom, adresse: adresse,
telephone: telephone);

  Adresse a3 = Adresse(numeroDeVoie: 15, nomDeLaVoie: "Avenue de la république",
codePostal: "44000", nomDeLaCommune: "Nantes");

  Personne u3 = Personne(prenom, nom: nom, adresse: a3, telephone: telephone);
}
```

Une instance d'`Adresse` est créée, elle s'appelle « `a3` ». Pour cela, le constructeur d'`Adresse` est mis à contribution. Tous les paramètres sont renseignés. Une fois, l'instance prête, il suffit de la fournir en paramètre au constructeur de `Personne`.

Dans le code ci-dessus, une personne ne possède qu'une seule adresse, il s'agit d'une relation unique. Parfois, elle est multiple. Dans le cas du téléphone, il n'est pas rare qu'une personne puisse posséder deux voire trois numéros, un fixe, un professionnel (bureau) et un portable. Il faudra alors gérer une collection comme une liste (ex. : `List<Telephone> listeNumerosDeTelephone`).

© Editions ENI - All rights reserved

3.8 Héritage

Au-delà de l'association qui consiste à faire travailler ensemble certains objets, il existe une autre notion tout aussi importante, l'héritage.

Comme son qualificatif le sous-entend, la relation est plus forte qu'une simple association. Dans ce contexte, l'héritage est utilisé quand des points communs existent entre deux ou plusieurs classes. Il est alors possible de « rassembler » ces traits de caractère communs dans une classe mère et d'en faire hériter les classes filles.

Pour continuer sur l'exemple de ce chapitre, il serait possible d'envisager que la classe `Personne` soit suffisamment généraliste pour devenir une classe mère. Descendant de celle-ci, un logiciel pourrait avoir besoin de gérer deux types de personnes : des utilisateurs et des salariés.

Ces deux catégories auraient en point commun tout ce qui est compris dans la classe Personne et possèderaient en plus des éléments propres. Un utilisateur pourrait avoir un identifiant et un mot de passe. Le salarié, lui, a un salaire.

Ceci peut être représenté par un diagramme :

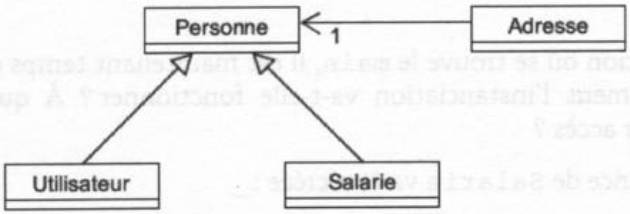


Figure 3 - Diagramme de classe

Côté code, pour établir le lien, il convient d'utiliser le mot-clé `extends` sur la classe fille. Pour `Utilisateur`, le résultat est :

```
import 'Personne.dart';

class Utilisateur extends Personne {
 String login;
 String motDePasse;
```

```
 Utilisateur(String prenom, this.login, this.motDePasse) :
super('prenom');

}
```

Après avoir procédé à l'importation de `Personne`, il est possible de l'ajouter en extension au nom de la classe :

```
class Utilisateur extends Personne {
```

Concernant les attributs, il n'y a pas de changement par rapport à d'habitude. En revanche, le constructeur évolue, il doit prendre en considération le constructeur de la classe mère. Le mot-clé `super` permet d'y faire référence.

```
Utilisateur(String prenom, this.login, this.motDePasse) :
super('prenom');
```

Pour la seconde classe, `Salarie`, le même travail est à réaliser :

```
import 'Personne.dart';

class Salarie extends Personne {
 double salaire;

 Salarie(String prenom, this.salaire) : super(prenom);

}
```

Dans le fichier d'exécution où se trouve le `main`, il est maintenant temps de tester tout cela. Comment l'instanciation va-t-elle fonctionner ? À quoi l'instance va-t-elle avoir accès ?

Pour débuter, une instance de `Salarie` va être créée :

```
Salarie s1 = Salarie('Paul', 1500);
```

Le constructeur appelé est celui présent dans la classe. Il attend le paramètre obligatoire du prénom ainsi que le montant du salaire.

À partir de ce salarié, l'accès à des éléments propres à la classe mère comme le nom ou le téléphone est maintenant autorisé.

```
s1.nom = "Durand";
s1.telephone = "0203040506";
```

© Editions ENI - All rights reserved

À travers cet exemple, il est évident que l'héritage permet l'accès à toutes les composantes de la classe mère. Par cette voie, un salarié possède lui aussi un prénom, un nom, un numéro de téléphone et une adresse. Au-delà, il peut avoir ses propres particularités, comme ici un salaire.

Par souci de vérification, il est envisageable d'utiliser `print` afin de voir si ces valeurs ont été pris en compte.

```
print("Le salarié s s'appelle : ${s1.prenom} ${s1.nom},
il gagne ${s1.salaire} par mois");
```

Dans la console, l'exécution de ce code produit le résultat suivant :

Le salarié s s'appelle : Paul Durand, il gagne 1500.0 par mois

Un autre avantage de l'héritage est le mécanisme appelé transtypage. Cela désigne le fait qu'une instance peut accepter plusieurs types, en l'occurrence celui de la classe fille évidement mais aussi celui de la classe mère.

```
Personne p1 = Salarie("John", 3000);
```

Dans cette situation, pour un développeur qui la manipulerait ultérieurement, il serait difficile de savoir si `p1` est une instance de `Personne`, d'`Utilisateur` ou de `Salarie`. Heureusement, il existe une fonction qui permet de le savoir `runtimeType()`.

```
print(p1.runtimeType);
```

Si cette fonction est appliquée à `p1`, la réponse donnée dans la console est très explicite :

Salarie

Il est également possible d'attribuer à une `Personne` une instance déjà créée de `Salarie`. Dans le code suivant, l'instance `s1` de `Salarie` est affectée à une personne nommée `p2` :

```
Personne p2 = s1;
```

L'inverse est aussi possible, mais présente un risque supplémentaire :

```
Salarie s2 = p1;
```

Dans le cadre d'un héritage, il est certain qu'un enfant se présente comme le parent. D'ailleurs, il n'en possède qu'un seul et unique puisque Dart ne supporte pas l'héritage multiple. Dans le sens opposé, il existe plusieurs enfants. Rien ne permet d'avoir l'assurance qu'il s'agisse bien du bon sauf en vérifiant :

```
if (p1.runtimeType == Salarie) {
 Salarie s3 = p1;
 print("Salaire de s3 : ${s3.salaire}");
}
```

Dans la console, `print` affiche :

Salaire de s3 : 3000.0

Si par mégarde, une mauvaise affectation est réalisée, une exception sera levée. Dans le code ci-dessous, `p1`, qui est un salarié, est affecté à un utilisateur s'appelant `u1`.

```
//Erreur de typage :
Utilisateur u1 = p1;
```

L'exécution de ce programme entraîne immédiatement la levée d'une exception affichée dans la console :

Unhandled exception:
type 'Salarie' is not a subtype of type 'Utilisateur'

3.9 Surcharge de méthodes

Dans le cadre de l'héritage, il est aussi possible de « surcharger » une méthode appartenant à la classe mère à l'intérieur d'une classe fille. Si une fonction `parler()` était ajoutée dans `Personne`, il serait possible de la réutiliser dans les classes filles. Deux possibilités sont offertes, soit la conserver et y ajouter du code, soit la modifier totalement pour la faire correspondre à un besoin précis. Il s'agit d'une surcharge de méthode.

© Editions ENI - All rights reserved

Voici la méthode dans la classe `Personne` :

```
String parler(String phrase) => "Une personne parle et dit $phrase";
```

À ce moment, la méthode est accessible grâce à l'héritage sur toutes les instances de `Salarie` et `Utilisateur` :

```
print(s1.parler("bonjour"));
```

Dans la console, le message qui apparaît est le suivant :

Une personne parle et dit bonjour

Dans la classe fille `Salarie`, elle peut donc être appelée en conservant le traitement d'origine grâce à l'utilisation du mot-clé `super` :

```
String parler(String phrase) {
 return super.parler(phrase) + ", il s'agit d'un salarié";
}
```

Le résultat de `print` sur l'appel de `parler()` est donc la combinaison des deux éléments :

Une personne parle et dit bonjour, il s'agit d'un salarié

L'autre possibilité est de modifier totalement le corps de la méthode pour l'adapter parfaitement au contexte :

```
String parler(String phrase) {
 return "Un salarié parle et dit $phrase";
}
```

Dans ce cas, il n'y a pas de confusion entre la méthode de la classe mère et celle de la classe fille au moment de l'appel. Le résultat affiché par la console le prouve :

Un salarié parle et dit bonjour

3.10 Interface

Dart ne tolère pas l'héritage multiple comme indiqué précédemment. C'est d'ailleurs le cas de nombreux langages. Pour pallier cela, il existe un autre type de classe appelée « interface ». Elle permet d'obliger une classe à passer un contrat, l'implémentation. Toutes les fonctions prévues dans l'interface devront être réécrites dans la classe qui l'implémente.

L'utilité est de venir compléter les effets de l'héritage. Quand des comportements en commun sont trouvés et qu'un héritage est déjà en place, l'interface peut prendre le relais.

Dans l'exemple courant, il convient de créer une nouvelle classe qui jouera le rôle d'interface.

```
abstract class Marcheur {
 void marcher();
}
```

Le plus souvent, cette classe qui ne sert qu'à exposer certaines méthodes sera qualifiée d'abstraite (`abstract`). C'est-à-dire qu'elle ne pourra pas être instanciée. Il n'y aurait aucun intérêt à le faire. Ainsi, il est impossible d'écrire ceci :

```
Marcheur m = Marcheur();
```

En revanche, elle conviendra parfaitement pour endosser le rôle d'interface. La méthode s'y trouvant ne comporte pas de code. Elle est juste énoncée.

Pour qu'une classe implémente l'interface, il faut utiliser `implements` suivi du nom de l'interface. Dans l'exemple, cette dernière sert à exposer une méthode nommée `marcher()` qui pourrait décrire la façon de se mouvoir de certaines catégories d'objets. Les classes `Utilisateur` et `Salarie` pourraient l'implémenter, car les deux ont en commun de marcher. Il aurait donc été possible d'intégrer cela à la classe mère. En revanche, si une classe `Chien` est créée plus tard, le fait d'avoir intégré la méthode `marcher()` à `Personne` compliquerait les choses puisqu'un chien marche également, mais ne peut pas hériter de `Personne` car les différences sont trop nombreuses.

L'interface est la solution à ce problème. Elle implique l'obligation de décrire `marcher()` sans pour autant imposer le poids d'un héritage.

© Editions ENI - All rights reserved

Sur la classe `Salarie`, l'implémentation donne ceci :

```
import 'Marcheur.dart';
import 'Personne.dart';

class Salarie extends Personne implements Marcheur{
 double salaire;

 Salarie(String prenom, this.salaire) : super(prenom);

 String parler(String phrase) {
   return "Un salarié parle et dit $phrase";
 }
 @override
 void marcher() {
   print("Je marche comme un salarié !");
 }

}
```

La première ligne de la classe décrit bien l'héritage de `Personne` et l'implémentation de l'interface `Marcheur` respectivement avec `extends` et `implements`.

```
class Salarie extends Personne implements Marcheur{
```

Une fois l'implémentation de l'interface réalisée, l'IDE signale qu'une méthode issue de cette dernière doit absolument être ajoutée. C'est le fameux contrat passé.

En insérant la méthode `marcher()`, une annotation `@override` apparaît indiquant au système qu'il s'agit d'une surcharge de méthode. L'IDE place par défaut un commentaire `TODO` à l'intérieur de la méthode. En d'autres termes, il indique qu'il reste du travail à faire et que du code doit être inséré à cet endroit afin d'obtenir un comportement correspondant à la situation. Après tout, ce n'est pas parce qu'une personne et un chien marchent, qu'ils le font de la même façon. L'humain est bipède, le chien marche à quatre pattes.

Le test de l'exécution du code est simple :

```
s1.marcher();
```

Le résultat est le suivant :

Je marche comme un salarié !

3.11 Mixin

Les Mixins sont une notion très particulière. Comme le nom l'indique, il s'agit d'un mix, d'un « mélange entre ». Entre quoi ? C'est une solution intermédiaire située entre l'héritage et l'interface.

Cela permet notamment à une classe qui possède déjà un héritage de pouvoir appeler des méthodes se trouvant dans une classe tierce sans avoir à les surcharger et de pouvoir les utiliser telles quelles sont.

Afin de rendre cela concret, il convient de pratiquer. Une classe `Profession` est mise en place :

```
class Profession {
  String metier;

  String competence(List<String> competences) {
    StringBuffer sb = StringBuffer();

    sb.writeln("Le $metier a les compétences suivantes : ");

    for(String c in competences) {
      sb.writeln("- $c");
    }

    return sb.toString();
  }
}
```

Cette classe possède un attribut `metier` et une méthode `competence` qui va servir, comme son nom l'indique, à détailler l'ensemble de ces dernières.

À l'intérieur, un objet `StringBuffer` est construit. Il a l'avantage de pouvoir assembler des chaînes de caractères et d'être plus performant que le type `String` dans ce cas d'usage.

© Editions ENI - All rights reserved

Dans la classe `Utilisateur`, il est possible d'ajouter le lien à la classe `Profession` comme ceci :

```
import 'Personne.dart';
import 'Profession.dart';

class Utilisateur extends Personne with Profession {
 String login;
 String motDePasse;

 Utilisateur(String prenom, this.login, this.motDePasse) : super('prenom');
}
```

Sur la première ligne, cette fois, c'est le mot `with` qui fait son apparition. Il vient en complément de l'héritage.

À la différence de l'interface qui obligeait à réécrire le code contenu dans la méthode, ici, ce n'est pas demandé. La méthode va pourvoir être utilisée dans sa forme originale. Dans l'exemple, il y avait un utilisateur u2 sur lequel le Mixin va être testé :

```
u2.metier = "Développeur";

List<String> competences = ["Flutter", "Dart", "Java", "c++"];

print(u2.competence(competences));
```

Grâce au Mixin, il est possible d'assigner une valeur à l'attribut `metier`. La méthode peut être appelée et elle s'exécute comme prévu selon le code mis en place dans `Profession`. Dans la console, il apparaît ceci :

Le Développeur a les compétences suivantes :
- Flutter
- Dart
- Java
- c++

Les Mixins sont un bon complément à l'héritage et aux interfaces. Ils offrent plus de souplesse sans pour autant basculer dans l'héritage multiple.

3.12 Extension

Dart est un langage vraiment complet qui offre un panel impressionnant de fonctionnalités. Parmi celles-ci, il en existe une qui est vraiment appréciable, il s'agit des extensions.

Elles permettent de rajouter du code, notamment des méthodes, au sein de classes dans un fichier à part. Ainsi, il n'y a pas besoin de modifier le fichier d'origine. Elles trouvent tout leur sens, dans l'utilisation d'API. Un code qui a souvent été produit par d'autres répond parfois partiellement aux besoins rencontrés. Dans ce cadre, la mise en place d'une extension peut s'avérer utile pour le compléter.

Dans la classe `Salarie`, il existe un `salaire`. Admettons que cette classe soit le fruit d'un autre développeur et qu'elle ait été récupérée. La problématique qui se pose est de pouvoir convertir le salaire en dollars. Or, cette fonctionnalité n'existe pas au sein de cette classe.

Plutôt que de modifier le code de `Salarie` directement, une extension peut résoudre le problème. L'extension `SalarieExtension` est créée :

```
import 'Salarie.dart';

extension SalarieExtension on Salarie {
 double convertirSalaireDollars() {
   return this.salaire * 1.11;
 }
}
```

Attention, il ne s'agit pas vraiment d'une classe. La première ligne est donc impactée par cela. La syntaxe correcte est du type : **`extension`** `nomDeLExtension` **`on`** `nomDeLaClasseAEtendre`. Dans la signature, il faut bien évidemment préciser la classe étendue grâce au mot-clé `on`.

Dans l'extension présentée ci-dessus, il n'existe qu'une seule fonction qui convertit le salaire d'euros en dollars. Il est particulièrement intéressant de noter que le lien établi par `on` permet l'utilisation et le travail sur les attributs de la classe étendue. Ici, le retour est calculé par la multiplication du salaire par le taux de 1,11 courant au moment de la rédaction de l'ouvrage. L'appel à l'attribut `salaire` passe par le mot-clé `this`.

© Editions ENI - All rights reserved

```
return this.salaire * 1.11;
```

Pour utiliser l'extension, il est toutefois nécessaire de réaliser un `import` sans lequel les rajouts ne seront pas accessibles. Mais ceci fait, il n'y a aucune limite à l'utilisation de la méthode sur une instance de `Salarie`.

```
import 'Personne.dart';
import 'Salarie.dart';
import 'Utilisateur.dart';
import 'SalarieExtension.dart';

main() {
 Salarie s1 = Salarie('Paul', 1500);

 print("Le salaire de s1 est de ${s1.convertirSalaireDollars().
toStringAsFixed(2)} dollars");


}
```

La fonction est placée dans l'argument de `print` afin d'obtenir un rendu dans la console. Préalablement, pour éviter d'afficher un nombre non contrôlé de décimales, la méthode `toStringAsFixed()` est appelée. Elle prend en paramètre un entier qui définit le nombre de chiffres souhaités après la virgule. L'exécution du code donne ceci :

Le salaire de s1 est de 1665.00 dollars

3.13 Static

Dans tous les exemples qui ont été vus jusqu'à présent, les méthodes écrites dans les classes ne peuvent être utilisées qu'à partir d'une instance qui existe. Les méthodes dites `static` dérogent à cette règle. Pour appeler une de ces méthodes, il suffit de s'appuyer sur le nom de la classe. Elles sont intégrables dans n'importe quelle classe. Pour l'exemple, une méthode `static` va être ajoutée à l'extension. Elle permettra de connaître le taux utilisé dans le cadre de la conversion Euro/Dollar vue plus haut. Ce traitement trouve tout son sens au travers d'une méthode `static` puisqu'il sera pratique de connaître le taux à n'importe quel instant et pas seulement après avoir instancié la classe.

Le code de l'extension doit subir quelques modifications. La valeur du taux de conversion est extraite dans une constante afin de devenir manipulable. Le code devient :

```
import 'Salarie.dart';

final double _tauxEuroDollar = 1.11;

extension SalarieExtension on Salarie {
 double convertirSalaireDollars() {
   return this.salaire * _tauxEuroDollar;
 }

 static double getTauxEuroDollard() {
   return _tauxEuroDollar;
 }
}
```

De plus, la méthode `getTauxEuroDollar()` est ajoutée avec le mot-clé `static`. Son appel est réalisé comme expliqué plus haut, directement sur le nom de la classe soit `SalarieExtension` :

```
import 'SalarieExtension.dart';

main() {

 print("Le taux Euro/Dollar est de : ${SalarieExtension.getTauxEuroDollard()}");

}
```

Dans l'exemple, il n'existe aucune instance. Malgré tout, il est possible d'exécuter la méthode. En console, le taux demandé apparaît :

Le taux Euro/Dollar est de : 1.11

© Editions ENI - All rights reserved

4. Généricité

La généricité est couramment utilisée pour rendre le travail plus concis et polyvalent. Elle offre la possibilité à une classe qui s'appuie sur une autre dans le cadre d'une association de faire en sorte de ne plus se limiter à celle-ci, mais de pouvoir accueillir n'importe quelle classe à la place.

La généricité a déjà été abordée sans le savoir durant le chapitre précédent au travers des listes. Les listes peuvent travailler avec tous types d'objets. La généricité se reconnaît par sa syntaxe singulière où le type attendu s'écrit entre chevrons.

```
List<int> listeDeNombres;
```

Il existe bien sûr quelques classes génériques déjà présentes dans le SDK comme les listes. Mais l'intérêt principal est de pouvoir créer nos propres classes usant de cet effet.

Dans l'exemple suivi dans ce chapitre, une classe `Adresse` avait été créée et associée à `Personne`. Cependant, en fonction des pays de résidence, les adresses varient. Si les personnes utilisant l'application sont françaises ou américaines, il faudra être capable de gérer les deux sortes d'adresses. La différence se situe principalement sur le fait qu'en France un code postal (de cinq caractères) est utilisé, alors qu'aux États-Unis il s'agit d'un code ZIP (sur cinq ou neuf caractères) additionné à un État.

Pour commencer, il faut modifier `Adresse` puisque cette classe ne sera plus instanciée. Elle va donc devenir `abstract` puis servira de classe mère à deux nouvelles classes : `AdresseFR` et `AdresseUS`. L'attribut `codePostal` va aussi disparaître.

Voici le code modifié de `Adresse` :

```
abstract class Adresse {
 int _numeroDeVoie;
 String _nomDeLaVoie;
 String _nomDeLaCommune;

 Adresse(int numeroDeVoie, String nomDeLaVoie,
     String nomDeLaCommune) {
   this._numeroDeVoie = numeroDeVoie;
   this._nomDeLaVoie = nomDeLaVoie;
```

```
    this._nomDeLaCommune = nomDeLaCommune;
  }

  String get nomDeLaCommune => _nomDeLaCommune;

  set nomDeLaCommune(String value) {
    _nomDeLaCommune = value;
  }

  String get nomDeLaVoie => _nomDeLaVoie;

  set nomDeLaVoie(String value) {
    _nomDeLaVoie = value;
  }

  int get numeroDeVoie => _numeroDeVoie;

  set numeroDeVoie(int value) {
    _numeroDeVoie = value;
  }
}
```

Il faut maintenant créer les deux classes filles. La première, AdresseFR, va gérer les adresses au format français et la seconde, AdresseUS, celles au format américain.

Voici le code de AdresseFR :

```
import 'Adresse.dart';

class AdresseFR extends Adresse{
 String codePostal;

 AdresseFR(this.codePostal, int numeroDeVoie, String nomDeLaVoie,
     String nomDeLaCommune ) : super(numeroDeVoie, nomDeLaVoie,
nomDeLaCommune);

}
```

© Editions ENI - All rights reserved

Et celui de AdresseUS :

```
import 'Adresse.dart';

class AdresseUS extends Adresse {
 String zipCode;
 String etat;

 AdresseUS(this.zipCode, this.etat, int numeroDeVoie, String
nomDeLaVoie, String nomDeLaCommune) : super(numeroDeVoie,
nomDeLaVoie, nomDeLaCommune);

}
```

Les deux classes héritent d'Adresse. Elles appellent le constructeur de la classe mère et rajoutent les paramètres des attributs en place.

L'adresse était associée à la classe Personne. Il va falloir rendre cette classe générique. Elle doit pouvoir travailler avec des adresses, peu importe le type, qu'elles soient américaines ou françaises. Dans la signature de la classe, la généricité vient s'ajouter :

```
class Personne<T> {
```

Cette syntaxe indique qu'un élément générique est attendu. Il sera utilisé après dans le code de la classe. Le seul inconvénient, est que T peut être remplacé par n'importe quel type. Cette situation n'est pas limitative. Or, on ne souhaite obtenir que des adresses. L'idée va consister à s'appuyer sur le point commun de tous les types d'adresse : la classe mère Adresse. Ainsi la syntaxe suivante est permise et restreint le type attendu aux classes filles d'Adresse:

```
class Personne<T extends Adresse> {
```

Dès lors, le reste du code doit s'adapter. Le symbole T, qui est le type générique, va remplacer toutes mentions d'Adresse dans le reste de la classe :

```
import 'Adresse.dart';

class Personne<T extends Adresse> {
 String nom;
 String _prenom;
 String telephone;
 //String adresse;
 T adresse;
```

```
//Un constructeur pour tous les attributs avec le prénom obligatoire
Personne(this._prenom, {this.nom, this.telephone, this.adresse});

//Un constructeur nommé etatCivil qui n'attend que le prénom et le nom
Personne.etatCivil(this._prenom, this.nom);

String get prenom => _prenom;

set prenom(String value) {
  _prenom = value;
}

@override
String toString() {
  return "La Personne s'appelle $nom $_prenom";
}

String parler(String phrase) => "Une personne parle et dit $phrase";

T getAdresse() {
  return adresse;
}

}
```

La première modification concerne l'attribut qui voit son type passer d'`Adresse` à `T`. Pour le reste, il n'y a pas de différences notables avec le code déjà fourni. Pour étayer l'utilisation du type générique, une méthode `getAdresse()` est ajoutée. Elle renverra une adresse, elle s'appuie donc sur `T` pour cela.

La dernière chose à examiner est l'instanciation. Comment travailler avec une classe générique ? L'utilisation dans `main` pourrait donner ceci :

```
import 'AdresseFR.dart';
import 'AdresseUS.dart';
import 'Personne.dart';
import 'Salarie.dart';
import 'Utilisateur.dart';
import 'SalarieExtension.dart';

main() {
 AdresseFR aFr = AdresseFR('44000', 110, 'rue de la Paix', "NANTES");

 AdresseUS aUs = AdresseUS('3302', 'CA', 150, "Hollywood", "Los Angeles");

 Personne<AdresseFR> p3 = Personne("John", nom: "Doeuf", adresse: aFr);
```

© Editions ENI - All rights reserved

```
  Personne<AdresseUS> p4 = Personne("John", nom: "Doe", adresse: aUs);

  print(p3.getAdresse().runtimeType);
  print(p4.getAdresse().runtimeType);

}
```

Au préalable, deux adresses sont créées (`aFr` et `aUs`) soit une pour chaque pays. Puis deux personnes sont instanciées avec le type attendu entre chevrons. Puisque ce type a été restreint plus haut, il n'est possible d'indiquer ici que des types héritant d'`Adresse` et rien d'autre. Pour tester, voici une tentative non conforme :

```
Personne<String> p5 = Personne("James");
```

L'exécution de ce code entraîne irrémédiablement une erreur affichée dans la console :

```
Error: Type argument 'String' doesn't conform to the bound 'Adresse' of
the type variable 'T' on 'Personne'.
 - 'Adresse' is from 'Adresse.dart'.
Try changing type arguments so that they conform to the bounds.
Personne<String> p5 = Personne("James");
                    ^

Personne.dart:3:16: Context: This is the type variable whose bound
isn't conformed to.
class Personne<T extends Adresse> {
               ^
```

Une fois le type correct posé, le constructeur de `Personne` est appelé. Attention, maintenant que le type a été défini entre les chevrons, il convient de ne pas se tromper dans l'adresse à fournir. Une inversion comme celle-illustrée ci-dessous provoquera aussi une erreur :

```
Personne<AdresseUS> p4 = Personne("John", nom: "Doe", adresse: aFr);
```

Le système attend une adresse américaine (le type indiqué entre les chevrons) et c'est une française (`aFr`) qui lui est envoyée. Dans la console, le message est explicite :

```
Error: The argument type 'AdresseFR' can't be assigned to the parameter
type 'AdresseUS'.
- 'AdresseFR' is from 'AdresseFR.dart'.
- 'AdresseUS' is from 'AdresseUS.dart'.
Personne<AdresseUS> p4 = Personne("John", nom: "Doe", adresse: aFr);
```

Pour finir, les deux dernières lignes de `main` vont afficher dans la console le type des adresses afin de vérifier que le processus fonctionne bien.

```
print(p3.getAdresse().runtimeType);
print(p4.getAdresse().runtimeType);
```

Le résultat est conforme à celui souhaité :

```
AdresseFR
AdresseUS
```

5. Exception

Les exceptions sont des erreurs. Elles peuvent se lever à divers moments pendant l'exécution du programme comme au milieu d'un traitement de type algorithmique ou bien à la suite d'une action de l'utilisateur. Elles indiquent que quelque chose s'est mal déroulé.

Elles peuvent se classer en deux catégories :

- Les exceptions dites « systèmes ». Elles sont dues, la majorité du temps, à une action courante et potentiellement dangereuse et pour laquelle une exception bien particulière a été prévue.
- Les autres sont plutôt liées à des problématiques que nous nous fixons nous-mêmes, comme des contraintes métier. Elles sont volontairement levées quand une action non conforme a lieu.

© Editions ENI - All rights reserved

5.1 Exceptions systèmes

Les exceptions systèmes sont courantes. Elles répondent aux problématiques les plus fréquentes. Dart comprend six grands types d'exceptions :

- `DefferredLoadException` : levée quand une bibliothèque différée ne se charge pas.
- `FormatException` : levée lorsqu'une chaîne de caractères ou d'autres données ne possèdent pas le format attendu et ne peuvent pas être converties ou traitées.
- `IntegerDivisionByZeroException` : levée quand une division par zéro est tentée sur un nombre.
- `IOException` : levée quand un problème survient lors des opérations de types IO (Input/Output).
- `IsolateSpawnException` : levée quand un `isolate` ne peut pas être créé.
- `TimeOutException` : levée lorsqu'un délai d'expiration planifié est atteint en attendant un résultat asynchrone.

Toutes ces exceptions implémentent la classe `Exception` qui est plus générique. Il faut également préciser que d'autres exceptions encore plus spécifiques existent. Ces dernières implémentent au moins une des six exceptions citées plus haut.

Voici un exemple afin de mettre cela en pratique. Le code suivant est incorrect :

```
void main() {
 String nom = null;

 nom.toUpperCase();
}
```

Il n'est pas possible d'appliquer `toUpperCase()` à un élément qui vaut `null` sous peine d'obtenir ce message :

```
Unhandled exception:
NoSuchMethodError: The method 'toUpperCase' was called on null.
Receiver: null
Tried calling: toUpperCase()
#0      Object.noSuchMethod (dart:core-patch/object_patch.dart:51:5)
#1      main (file:///E:/ProjetDart/ExceptionMain.dart:4:7)
#2      _startIsolate.<anonymous closure>
(dart:isolate-patch/isolate_patch.dart:305:19)
#3      _RawReceivePortImpl._handleMessage
(dart:isolate-patch/isolate_patch.dart:172:12)
```

Ce message apparaît subitement dans la console et démontre que ce code n'était pas verrouillé face à ce type de problème.

Quand une opération apparait dangereuse, il convient de placer le code à l'intérieur d'un bloc `try`. Si une erreur arrive, elle sera captée dans un second bloc, `catch`. Pour le code ci-dessus, cela implique la correction suivante :

```
void main() {
  String nom = null;

  try {
    nom.toUpperCase();
  } catch (e) {
    print("Problème sur toUpperCase : ${e.toString()}");
  }
}
```

L'action risquée est l'appel à `toUpperCase()`. Un bloc `try` est donc inscrit autour. L'exception, si elle est déclenchée, sera captée dans le bloc `catch`. Le message sera affiché dans la console.

```
Problème sur toUpperCase : NoSuchMethodError:
The method 'toUpperCase' was called on null.
Receiver: null
Tried calling: toUpperCase()
```

© Editions ENI - All rights reserved

Au premier abord, cela ne semble pas changer grand-chose puisqu'un message quasi-identique au précédent apparaît dans la console. La différence réside dans le fait que cette exception est contrôlée. C'est bien le développeur qui choisit quoi en faire.

Le but est potentiellement d'avertir l'utilisateur du problème ou bien d'alimenter un journal. Pour autant, le message va devoir être transmis du fichier où l'exception s'est levée jusqu'au fichier désiré comme une interface utilisateur. Pour réaliser cela, il va falloir propager l'exception. À cet effet, le mot-clé `throw` existe.

Pour illustrer cela, il convient d'ajouter une fonction à la classe `Personne` :

```
String getNomEnMajuscule() {
 return nom.toUpperCase();
}
```

Cette fonction va uniquement renvoyer le nom de la personne en majuscule. Si le nom possède une valeur `null` alors l'exception sera levée ici même si cette fonction a été appelée ailleurs. Voici son appel dans `main` :

```
import 'Personne.dart';

void main() {

 Personne p1 = Personne("Julien");

 print(p1.getNomEnMajuscule());

}
```

Une instance de `Personne` est créée. Elle comporte un prénom, mais pas de nom. Pour rappel, les attributs qui ne sont pas renseignés au moment de l'instanciation prennent la valeur `null`. De fait, l'appel à `getNomEnMajuscule()` entraîne la levée d'une exception et le message d'erreur qui en découle :

```
Unhandled exception:
NoSuchMethodError: The method 'toUpperCase' was called on null.
Receiver: null
Tried calling: toUpperCase()
#0      Object.noSuchMethod (dart:core-patch/object_patch.dart:51:5)
#1      Personne.getNomEnMajuscule
(file:///E:/ProjetDart/Personne.dart:41:16)
#2      main (file:///E:/ProjetDart/ExceptionMain.dart:15:12)
#3      _startIsolate.<anonymous closure>
(dart:isolate-patch/isolate_patch.dart:305:19)
#4      _RawReceivePortImpl._handleMessage
(dart:isolate-patch/isolate_patch.dart:172:12)
```

Il est clairement indiqué sur la ligne #1 que l'exception provient de la fonction `getNomEnMajuscule()` de la classe `Personne`.

Pour éviter cela, le code va être modifié pour faire en sorte que si une exception se lève, alors elle sera propagée :

```
String getNomEnMajuscule() {
 if (nom != null) {
   return nom.toUpperCase();
 } else {
   throw 'LE NOM NE DOIT PAS ETRE NULL !!';
 }
```

L'exception levée sera propagée dans `main` et captée :

```
try {
 print(p1.getNomEnMajuscule());
} catch (e) {
 print(e);
}
```

© Editions ENI - All rights reserved

Dans la console, le résultat est net :

LE NOM NE DOIT PAS ETRE NULL !!

Enfin, il existe un dernier mot-clé : `finally`. Il vient s'ajouter dans un ultime bloc après celui de `catch` et sert à exécuter un traitement, peu importe qu'une exception se soit levée ou non.

```
try {
 print(p1.getNomEnMajuscule());
} catch (e) {
 print(e);
} finally {
 print("Je m'exécute quoiqu'il arrive !");
}
```

Pour donner un exemple précis dans lequel `finally` prend tout son sens, la fermeture d'une connexion à une base de données pourrait être citée. En effet, si une fois connecté, une opération fait planter l'application alors la connexion à la base reste active. Elle crée un embouteillage inutile qu'il faut éviter.

5.2 Exceptions personnalisées

Grâce aux notions d'héritage vues précédemment, il est possible de créer des exceptions personnalisées. Celles-ci vont être utiles dans tous les cas qui ne sont pas prévus par défaut avec les exceptions « système ». C'est notamment le cas, des erreurs dites « métier ». Il s'agit de contraintes de la vie courante par exemple, ou qui naissent d'une problématique bien précise dans l'utilisation de l'application.

Un exemple clair est celui du code postal qui en France doit absolument comporter cinq caractères. Pour une application qui collecterait ce genre de données, il sera primordial de vérifier que la saisie de l'utilisateur correspond bien à cette attente.

Pour cela, puisqu'il n'existe pas d'exception pour ce cas précis, il convient d'en créer une.

```
class ExceptionPerso implements FormatException {
 int _source;
 ExceptionPerso(this._source);

 @override
 String get message => "Le code postal doit comporter 5 caractères !";

 @override
 int get offset => 5-_source;

 @override
 get source => _source;
}
```

Cette classe va donc hériter de `FormatException`. Elle peut comporter des éléments variés tels que des attributs, des méthodes.

Le choix de ce type d'exception est plutôt logique en l'occurence puisqu'il s'agit bien potentiellement d'un problème de format de chaîne de caractères qui va être rencontré.

Dans l'exemple ci-dessus, en implémentant une `FormatException`, deux méthodes ainsi qu'un *getter* sont imposés.

Afin de configurer parfaitement cette exception personnalisée, un attribut privé nommé `_source` de type entier est créé et accompagné d'un constructeur qui permettra son affectation. `_source` correspond aux nombre de caractères contenus dans l'objet `String` qui lèvera l'exception.

La fonction `message` renvoie uniquement une information générale relative à l'exception levée.

Le *getter* `source` permet d'accéder à la valeur de l'attribut du même nom.

La fonction `offset` permet de calculer l'écart entre la chaîne de caractères récupérée et celle attendue.

© Editions ENI - All rights reserved

Dans la classe AdresseFr, le code est modifié pour devenir ceci :

```
import 'Adresse.dart';
import 'ExceptionPerso.dart';

class AdresseFR extends Adresse{
 String _codePostal;

 AdresseFR(String codePostal, int numeroDeVoie, String
nomDeLaVoie,
      String nomDeLaCommune ) : super(numeroDeVoie, nomDeLaVoie,
nomDeLaCommune) {
   this.codePostal = codePostal;
 }

 String get codePostal => _codePostal;

 set codePostal(String value) {
   if (value.length == 5) {
     _codePostal = value;
   } else {
     throw ExceptionPerso(value.length);
   }
 }
}
```

L'attribut codePostal devient privé. La mise en place de *getter/setter* est alors possible. Le constructeur oblige le passage par le *setter* où la vérification du nombre de caractères s'applique.

Dans ce dernier, une condition est posée. Si la valeur récupérée ne fait pas cinq caractères, une exception de type ExceptionPerso est propagée. Ici, c'est bien le constructeur de cette classe qui est appelé. Il est donc possible de lui transmettre le nombre de caractères de la valeur reçue.

Dans main, il ne reste plus qu'à tester en créant une adresse au format erroné :

```
try {
 Adresse a1 = AdresseFR('111', 1, 'Rue de la paie', 'NANTES');
} catch (e) {
 print(' ${e.message} La valeur saisie fait ${e.source}
caractères. Il manque ${e.offset} caractères.');
}
```

L'appel au constructeur d'`AdresseFR` est entouré d'un `try-catch`. La valeur du code postal est « *111* ». Elle n'est donc pas conforme aux attentes. L'exception personnalisée est levée dans le *setter* `codePostal` et propagée pour être récupérée dans le bloc `catch`.

Ce dernier permet d'afficher, dans la console, les diverses informations prévues par la classe, à savoir le message, le nombre de caractères de la valeur saisie et enfin le différentiel avec la valeur réellement attendue :

Le code postal doit comporter 5 caractères ! La valeur saisie fait 3 caractères.
Il manque 2 caractères.

5.3 Exceptions multiples

Il est aussi possible que plusieurs types d'exception soient levés sur une même portion de code. Dans ce cas, il est conseillé de mettre en place plusieurs `catch` qui viendront capter les différentes exceptions afin de leur appliquer un traitement différencié. La syntaxe est la suivante :

```
try {
 Adresse a1 = AdresseFR('111', 1, 'Rue de la paie', 'NANTES');
} on ExceptionPerso catch (e) {
 print(' ${e.message} La valeur saisie fait ${e.source}
caractères. Il manque ${e.offset} caractères.');
} catch  (e) {
 print("Autre exception");
}
```

Pour préciser le type d'exception que l'on souhaite capter, le mot-clé on suivi du nom de l'exception sont utilisés. Cette configuration peut être démultipliée autant de fois que nécessaire.

© Editions ENI - All rights reserved

6. Traitement asynchrone

L'asynchronicité est la possibilité pour une tâche d'être réalisée en parallèle d'une autre dans un laps de temps qui diffère. Par exemple, obtenir de l'information depuis une base de données peut s'avérer long. Dans cette situation, il faut à tout prix éviter de bloquer l'application, le reste du programme doit continuer à fonctionner pendant cette recherche. La sollicitation de la base sera donc asynchrone.

Dart dispose de deux mots-clés pour l'asynchronicité : `async` et `await`.

Conjointement, ils sont souvent associés à la classe `Future<T>`. Cette classe est utilisée pour représenter une réussite comme le résultat d'un calcul, ou la récupération de données (ou un échec), qui sera disponible à un moment précis dans le futur.

Il existe aussi une autre classe, `Stream`, qui permet de travailler sur le sujet. Cette notion sera abordée dans le chapitre Les États.

Pour illustrer les propos évoqués plus haut, mettons en place ce code :

```
Future<void> main() async {
 print("1- J'arrive en premier");
 print(await retour1());
 print("5- J'arrive en dernier");
}

Future<String> retour1()  {
 print("2- Je dois arriver en second");
 var retour = Future.delayed(Duration(seconds: 5), () =>
"3- J'arrive un peu en retard");
 print("4- Je dois arriver en quatrième position");
 return retour;
}
```

Une première méthode `main`, point d'entrée du programme, est considérée comme une méthode de type `Future`. À l'intérieur trois actions sont inscrites, des appels à `print` pour être précis. La seconde renvoie à une fonction `Future<String> retour1()` qui prévoie également deux affichages dans la console ainsi que la création d'une réponse de type `Future<String>` réalisée après un délai de 5 secondes. Le délai est mis en place grâce à la fonction `Future.delayed()` qui prend en premier paramètre un objet de type `Duration` puis prévoit le retour.

Sur `main`, il est important d'indiquer que de l'asynchronisme va exister. Le mot-clé `async` est donc placé dans la signature :

```
Future<void> main() async {
```

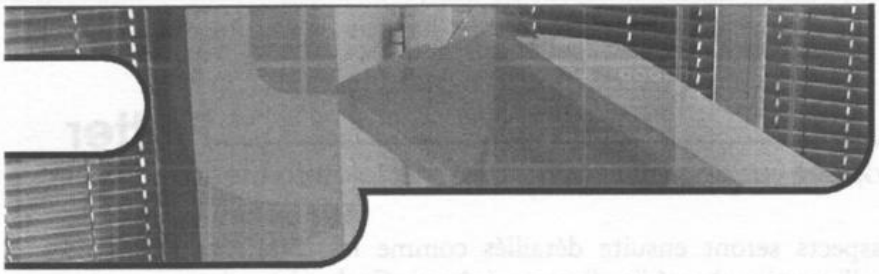
Puis, sur la fonction qui risque potentiellement de prendre du temps, il faut placer `await` pour indiquer « d'attendre » sa réponse.

```
print(await retour1());
```

Une fois le programme exécuté, dans la console, le résultat est intéressant. Le premier `print` apparaît immédiatement. Puis `retour1()` est appelée. Dès lors, le `print` n°2 s'affiche aussitôt. Le troisième, lui, mettra cinq secondes à se construire, il est en *standby*. Le programme continue donc son exécution et lit le quatrième message sans attendre que le précédent soit prêt. Enfin, avant de retourner dans `main`, il faudra attendre que le délai soit écoulé (c'est le rôle du `wait`), pour que la phrase n°3 apparaisse suivie immédiatement de la cinquième et dernière :

1- J'arrive en premier
2- Je dois arriver en second
4- Je dois arriver en quatrième position
3- J'arrive un peu en retard
5- J'arrive en dernier

© Editions ENI - All rights reserved



Chapitre 6
Les widgets de base

1. Introduction

Flutter est composé de widgets. C'est le cœur du framework dont le slogan est « *In Flutter everything is a Widget* » ! En français, on pourrait dire qu'avec Flutter tout est widget.

Le Widget est donc l'élément central du SDK. Il en existe pour à peu près tout, en allant des éléments graphiques jusqu'aux fonctionnalités.

À ce jour, sur la page web qui référence l'ensemble des Widgets, https://flutter.dev/docs/reference/widgets, il est possible d'en dénombrer 154.

L'objectif n'est pas de tous les détailler. Le chapitre va se consacrer à la découverte des principaux widgets pouvant servir lors d'un projet. D'autres seront abordés plus tard puisqu'ils possèdent certaines spécificités.

Tout d'abord, nous prendrons le temps d'examiner les structures graphiques principales sur les deux plateformes Android et iOS. Ce sera également l'occasion d'évoquer la détection de ces dernières pour obtenir un rendu adapté.

D'autres aspects seront ensuite détaillés comme les différents styles de boutons ou l'insertion de médias (images, icônes). Ce dernier point permettra aussi de s'attarder sur les dépendances et leur gestion. La découverte se poursuivra sur les textes et se finira sur les widgets servant à gérer la disposition d'autres widgets.

2. Material Component

Parmi les regroupements de widgets, il y a tous ceux qui concernent l'aspect graphique calé sur le style Material Design de Google. C'est celui qui est appliqué sur Android. Il conditionne la tenue générale de l'écran ainsi que d'autres éléments comme les boutons, par exemple.

2.1 Scaffold

Le `Scaffold` est la structure de base des applications Android. Il s'agit d'un ensemble pouvant regrouper plusieurs éléments comme une `appBar`, un `body`, un `floatingActionButton` ou encore une `bottomNavigationBar`.

Comme il s'agit d'un élément principal, il sera le premier à être retourné dans le widget de construction graphique, le `build`.

```
@override
Widget build(BuildContext context) {
 return Scaffold(
) ; //Scaffold
```

2.1.1 AppBar

L'`appBar` est un widget qui vient se situer en haut de l'écran et dans lequel le titre se trouve généralement placé. C'est un élément du `Scaffold`.

Il s'agit d'un outil complet qui peut également accueillir des boutons.

© Editions ENI - All rights reserved

Voici un exemple de code plaçant une appBar au sein d'un `Scaffold` et intégrant un bouton « j'aime ».

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     actions: <Widget>[
       IconButton(
           icon: _coeur,
           onPressed: _likeThis
       ),
     ],
     backgroundColor: Colors.red,
   ),
 );
}
```

Dans l'exemple, à l'intérieur de la structure basique, `Scaffold`, une `appBar` est intégrée. Elle comporte le titre de la page ainsi que le bouton « *j'aime* ». Pour varier un peu, la propriété `backgroundColor` est modifiée pour afficher la couleur rouge.

Pour le bouton, le choix se porte sur un widget IconButton. Il est intégré dans un paramètre nommé `actions` qui attend une liste de widgets. L'IconButton sera vu, peu après, plus en détail. La propriété `onPressed` appelle la méthode `_likeThis` :

```
void _likeThis() {
 setState(() {
   if (_likeBool) {
     _coeur = Icon(Icons.favorite_border, color: Colors.white,);
     _likeBool = false;
   } else {
     _coeur = Icon(Icons.favorite, color: Colors.white,);
     _likeBool = true;
   }
 });
}
```

Cette méthode incorpore un appel à `setState()`. Elle est donc à l'écoute de changements. En fonction d'un booléen qui surveille l'état de l'icône, ce dernier change pour passer d'un cœur vide à un cœur rempli. La déclaration de ces variables est réalisée précédemment comme ceci :

```
Icon _coeur = Icon(Icons.favorite_border, color: Colors.white,);
bool _likeBool = false;
```

Le code complet est le suivant :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo Home Page'),
   );
 }
}
class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);


 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
 Icon _coeur = Icon(Icons.favorite_border, color: Colors.white,);
 bool _likeBool = false;


 void _likeThis() {
   setState(() {
     if (_likeBool) {
```

© Editions ENI - All rights reserved

```
        _coeur = Icon(Icons.favorite_border, color: Colors.white,);
        _likeBool = false;
      } else {
        _coeur = Icon(Icons.favorite, color: Colors.white,);
        _likeBool = true;
      }
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
        actions: <Widget>[
          IconButton(
              icon: _coeur,
              onPressed: _likeThis
          ),
        ],
        backgroundColor: Colors.red,
      ),
    );
  }
}
```

Enfin, le rendu est celui-ci :

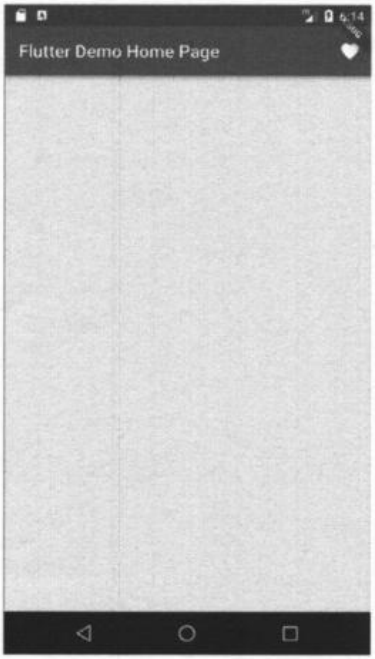


Figure 1 - AppBar avec un bouton

2.2 floatingActionButton

Le `floatingActionButton` est un widget déjà aperçu dans l'application du chapitre Ma première application. Il consiste en un bouton flottant situé, par défaut, en bas à droite de l'écran. Il existe un widget qui permet de gérer son emplacement, `floatingActionButtonLocation`.

Ce bouton prend au minimum une propriété `onPressed` qui désigne une méthode à appeler lors de l'appui sur celui-ci.

Généralement, un objet désigné par `child` lui est aussi associé. Le plus souvent, il s'agit d'une icône. Il existe d'autres propriétés comme `backgroundColor`.

© Editions ENI - All rights reserved

Pour repartir de l'écran précédent, il serait possible d'ajouter le code suivant :

```
floatingActionButton: FloatingActionButton(
 onPressed: _likeThis,
 backgroundColor: Colors.red,
 child: _coeur
),
```

Le bouton appelle la même méthode `_likeThis` que le bouton de l'`appBar`. Il incorpore aussi une icône en forme de cœur. Pour harmoniser le tout, le fond est coloré en rouge. Suite à l'appui sur le bouton, les deux cœurs se remplissent simultanément :

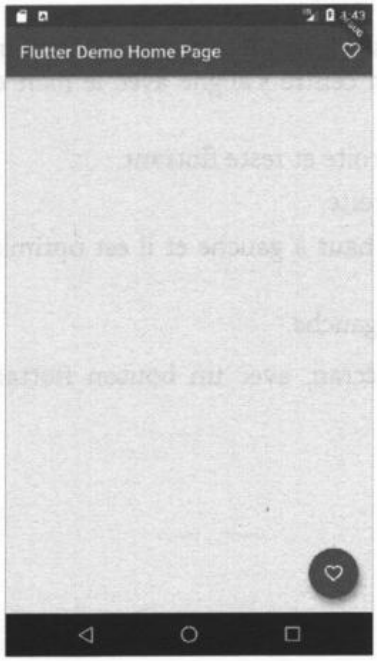


Figure 2 - Mise en place d'un floatingActionButton

2.3 FloatingActionButtonLocation

Attention, il s'agit bien d'un élément du `Scaffold` et non du `floatingActionButton`. Il est donc placé à part de celui-ci.

Il peut accepter les valeurs suivantes :

- `centerDocked` : le bouton flottant est aligné au centre. Il reste flottant sur une éventuelle `bottomNavigationBar`. Son centre s'aligne avec le haut de cette barre.
- `centerFloat` : le bouton flottant est centré et reste flottant en bas de l'écran.
- `endDocked` : le bouton flottant est en bas à droite. Il reste flottant sur une éventuelle `bottomNavigationBar`. Son centre s'aligne avec le haut de cette barre.
- `endFloat` : le bouton est situé en bas à droite et reste flottant.
- `endTop` : le bouton est aligné en haut à droite.
- `miniStartTop` : le bouton est placé en haut à gauche et il est optimisé pour les mini-boutons flottants.
- `startTop` : le bouton est situé en haut à gauche.

Voici un exemple, toujours sur le même écran, avec un bouton flottant centré :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     actions: <Widget>[
       IconButton(
           icon: _coeur,
           onPressed: _likeThis
       ),
     ],
     backgroundColor: Colors.red,
   ),
   floatingActionButton: FloatingActionButton(
       onPressed: _likeThis,
     backgroundColor: Colors.red,
     child: _coeur
```

© Editions ENI - All rights reserved

```
    ),
    floatingActionButtonLocation: FloatingActionButtonLocation.centerDocked,
  );
}
```

Ici, le code ne présente rien de particulier, hormis le fait que `floatingActionButtonLocation` est bien situé à la racine du `Scaffold` et non pas dans le `floatingActionButton` lui-même.

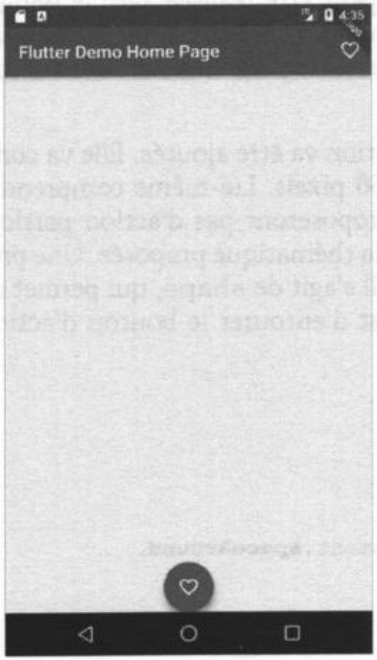


Figure 3 - Bouton d'action flottant centré grâce à floatingActionButtonLocation

2.4 bottomNavigationBar

La bottomNavigationBar est une barre de navigation située en bas de l'écran. Son implémentation peut être envisagée de plusieurs façons.

2.4.1 BottomAppBar

Dans un premier temps, l'implémentation peut être réalisée avec le widget BottomAppBar. Par défaut, sa taille est nulle. Il convient d'y inclure un Container par exemple, dont il est possible de déterminer la hauteur. Évidemment, la couleur est modifiable.

Dans l'exemple courant, une barre de navigation va être ajoutée. Elle va comprendre un Container d'une hauteur de 50 pixels. Lui-même comprendra deux IconButton qui pour l'instant ne proposeront pas d'action particulière. Cette barre sera rouge pour rester dans la thématique proposée. Une propriété supplémentaire est toutefois ajoutée. Il s'agit de shape, qui permet de définir une forme. En l'occurrence, l'idée est d'entourer le bouton d'action flottant placé précédemment.

```
bottomNavigationBar: BottomAppBar(
 color: Colors.red,
 child: Container(
   height: 50,
   child: Row(
     mainAxisAlignment: MainAxisAlignment.spaceAround,
     children: <Widget>[
       IconButton(
         icon: Icon(
           Icons.add_location,
           color: Colors.white,
         ),
         onPressed: null,
       ),
       IconButton(
           icon: Icon(
             Icons.forward,
             color: Colors.white,
           ),
         onPressed: null,
       ),
```

© Editions ENI - All rights reserved

```
      ],
    ),
  ),
  shape: CircularNotchedRectangle(),
),
```

Dans le constructeur de `BottomAppBar`, le paramètre `shape` prend la valeur `CircularNotchedRectangle`. Cette classe ne possède que ce constructeur public et rien d'autre. Elle permet d'obtenir un rectangle contenant une encoche circulaire dans laquelle le bouton d'action flottant va venir se loger.

Le résultat est le suivant :

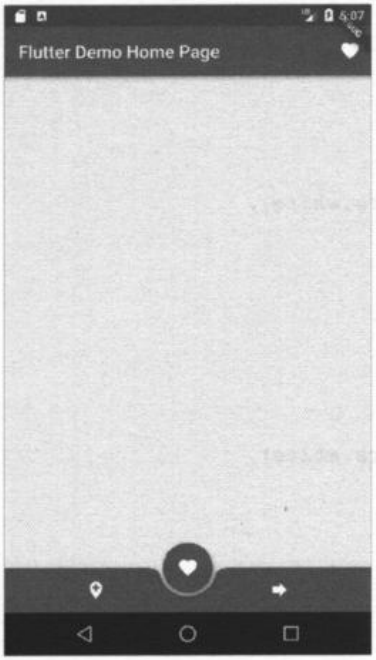


Figure 4 - Exemple de BottomNavigationBar

2.4.2 BottomNavigationBar

Une autre façon de travailler avec la barre est de l'implémenter avec `BottomNavigationBar`. Cette dernière comprendra des `BottomNavigationBarItem` permettant la navigation.

Pour rester dans l'exemple, la barre précédente (`BottomAppBar`) est supprimée ainsi que le positionnement centré du `floatingActionButton`.

Une fois les deux suppressions opérées, le code suivant est ajouté :

```
bottomNavigationBar: BottomNavigationBar(
 items: const <BottomNavigationBarItem>[
   BottomNavigationBarItem(
       icon: Icon(
         Icons.home,
         color: Colors.white,
       ),
       title: Text(
         'Accueil',
         style: TextStyle(color: Colors.white),
       )),
   BottomNavigationBarItem(
       icon: Icon(
         Icons.supervised_user_circle,
         color: Colors.white,
       ),
       title: Text(
         'Mon compte',
         style: TextStyle(color: Colors.white),
       )),
   BottomNavigationBarItem(
       icon: Icon(
         Icons.assessment,
         color: Colors.white,
       ),
       title: Text(
         'Statistiques',
         style: TextStyle(color: Colors.white),
       )),
 ],
 backgroundColor: Colors.red,
 onTap: _itemClique,
 currentIndex: _indexSelectionne,
),
```

© Editions ENI - All rights reserved

La première propriété de la BottomNavigationBar est items. Une collection de BottomNavigationBarItem y est alors renseignée :

```
bottomNavigationBar: BottomNavigationBar(
  items: const <BottomNavigationBarItem>[
    item 1,
    item 2,
    ...
  ])
```

Puis chaque item est alimenté par une icône ainsi qu'un texte simple. Pour se conformer à l'aspect graphique, la propriété backgroundColor est appelée ensuite. Elle prend une valeur correspondant au rouge.

La propriété currentIndex permet de définir sur quel item l'application se positionne. Quand un item est sélectionné, son style évolue. Le texte apparaît plus gros par exemple. Il faut noter que la numérotation des index des items commence à 0. Au démarrage de l'application, currentIndex doit donc valoir 0. Cependant, en fonction des clics, sa valeur évoluera. Il convient donc de créer une variable à mettre à jour : _indexSelectionne.

```
int _indexSelectionne = 0;
```

La mise à jour de l'index va avoir lieu à chaque fois qu'un des items sera cliqué. Pour cela la propriété onTap va appeler une méthode nommée _itemClique.

```
void _itemClique(int index) {
 setState(() {
   _indexSelectionne = index;
 });
}
```

Afin de vérifier que tout ceci fonctionne correctement, un texte est ajouté à la partie centrale, représentée par le paramètre body.

```
body: Center(
 child: Column(
   mainAxisAlignment: MainAxisAlignment.center,
   children: <Widget>[
     Text(
       '${_affichage}',
       style: TextStyle(color: Colors.red, fontSize: 40),
     ),
```

```
    ],
  ),
),
```

Ce texte, pour avoir une meilleure présentation, est inséré à l'intérieur de deux Widgets, `Center` et `Column`. La donnée du texte correspond à la valeur d'une variable nommée `_affichage` qui va changer en fonction de l'item choisi. Par défaut, au démarrage, elle doit être déclarée et initialisée comme suit :

```
String _affichage = "0: Accueil";
```

Il reste maintenant à modifier la méthode `_itemClique` pour faire évoluer le texte. Elle devient donc :

```
void _itemClique(int index) {
  setState(() {
    _indexSelectionne = index;
    switch (_indexSelectionne) {
      case 0:
        {
          _affichage = '$_indexSelectionne: Accueil';
        }
        break;
      case 1:
        {
          _affichage = '$_indexSelectionne: Mon compte';
        }
        break;
      case 2:
        {
          _affichage = '$_indexSelectionne: Statistiques';
        }
        break;
    }
  });
}
```

© Editions ENI - All rights reserved

Pour gérer les divers cas, une structure `Switch/Case` est mise en place. Dans chaque situation, l'index sera affiché accompagné de la catégorie. Après avoir cliqué sur l'item du milieu, l'aspect visuel est celui-ci :

Figure 5 - BottomNavigationBar

2.5 Drawer

Le `Drawer` est un menu caché sur la gauche qui glisse horizontalement. Il est surtout utilisé pour ne pas surcharger l'écran avec divers liens de navigation par exemple, et pour pouvoir les regrouper dans une zone dédiée et accessible uniquement en cas de besoin. Pour y avoir accès, l'`appBar` est complétée automatiquement par une icône « *hamburger* » qui correspond à trois traits horizontaux et parallèles.

Voici un exemple pour illustrer la mise en place d'un `Drawer` :

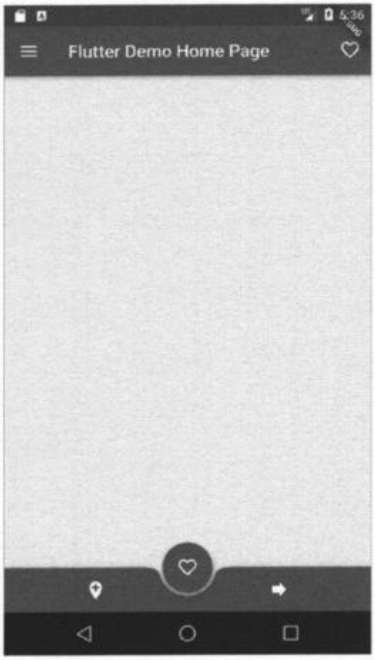
```
drawer: Drawer(
 child: Column(
   children: <Widget>[
     DrawerHeader(
       child: Text(
         'Menu de navigation',
         style: TextStyle(
             color: Colors.red,
           fontWeight: FontWeight.bold,
           fontSize: 25
         ),
       ),
     ),
     Text('Lien 1'),
     Text('Lien 2'),
     Text('Lien 3'),
   ],
 ),
),
```

Dans le code, un widget `Column` est privilégié. Toutefois, il est plutôt recommandé d'utiliser un widget `ListView`, puis des `ListTiles`. Comme ces notions seront vues plus tard au chapitre Listes et grilles, le choix a été fait de déroger un peu à la règle et d'employer un élément connu comme `Column`.

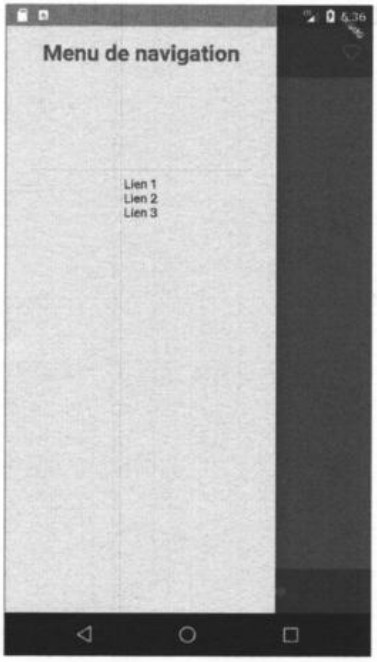
À l'intérieur, un widget `DrawerHeader` est d'abord mis en place. Il correspond à l'en-tête du `Drawer` qui contient généralement un titre voire des informations sur l'utilisateur. Puis, trois lignes de texte sont ajoutées pour simuler des liens de navigation vers d'autres écrans.

© Editions ENI - All rights reserved

Voici le rendu avant et après l'ouverture du `Drawer` :



Figures 5 - Mise en place d'un Drawer (fermé)



Figures 6 - Mise en place d'un Drawer (ouvert)

3. Cupertino (Style iOS)

En parallèle du style Google, le Material Design, qui s'applique à la plateforme Android, Flutter comprend des widgets pour iPhone correspondant au style présent sur iOS. Il s'agit des widgets Cupertino (nom donné en référence à la ville où se situe le siège d'Apple). Dans cette partie, tous les widgets Cupertino ne seront pas vus. Certains éléments plus spécifiques tels que les boutons seront abordés un peu plus loin.

© Editions ENI - All rights reserved

3.1 Scaffold de base

Sous iOS, il existe également un `Scaffold`. Il sert, tout comme celui du Material Design, à structurer l'écran. Il s'agit du `CupertinoPageScaffold`.

Afin de l'utiliser, il faut importer la bibliothèque Cupertino car, par défaut, seulement celle de Material est présente.

```
import 'package:flutter/cupertino.dart';
```

La suite du code est assez semblable au `Scaffold` vu plus haut :

```
@override
Widget build(BuildContext context) {
 return CupertinoPageScaffold(
     child: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[

         ],
       ),
     ),
   backgroundColor: Colors.red[100],
 );
}
```

L'exemple reprend la présentation vue couramment, à savoir un `Center` suivi d'un `Column`. Une propriété `backgroundColor` existe également pour définir la couleur de fond. L'autre widget pouvant être incorporé est une `navigationBar`. Avant de voir cet élément en détail, il existe un autre `Scaffold` à découvrir qui en possède déjà une.

3.2 Scaffold avec barre de navigation

Il existe un autre Widget correspondant au `Scaffold` mais qui gère en plus une barre de navigation cette fois-ci : le `CupertinoTabScaffold`.

Traditionnellement sur iOS, il existe une barre de navigation située en bas de l'écran. Globalement, le principe reste le même que précédemment. Malgré tout, il y a une particularité dans la création de la `CupertinoTabBar`. Bien qu'elle possède une propriété `items` comme la barre Material Design `BottomNavigationBar`, il faut incorporer en plus une propriété `tabBuilder` afin de définir les comportements appliqués à chaque item.

La structure de base est donc celle-ci :

```
@override
Widget build(BuildContext context) {
 return CupertinoTabScaffold(
   tabBar: CupertinoTabBar(
     items:
   ),
   tabBuilder: (context, index) {
   },
 );
}
```

Pour la propriété `items` de la `CupertinoTabBar`, le choix logique est de se tourner vers le widget `BottomNavigationBarItem`. La construction est strictement identique à la section BottomNavigationBar de ce chapitre.

```
tabBar: CupertinoTabBar(
 items: const <BottomNavigationBarItem>[
   BottomNavigationBarItem(
     icon: Icon(CupertinoIcons.home, color: Colors.white),
     title: Text('Accueil', style: TextStyle(color: Colors.white)),
   ),
   BottomNavigationBarItem(
     icon: Icon(CupertinoIcons.group, color: Colors.white),
     title: Text('Mon compte', style: TextStyle(color: Colors.white)),
   ),
   BottomNavigationBarItem(
     icon: Icon(CupertinoIcons.book, color: Colors.white),
     title: Text('Statistiques', style: TextStyle(color: Colors.white)),
   ),
 ],
```

© Editions ENI - All rights reserved

```
  backgroundColor: Colors.red,
),
```

La propriété `tabBuilder` présente une particularité. Elle travaille à partir de deux éléments : `context` et `index`. C'est ce deuxième paramètre qui va avoir un intérêt pour savoir quel item est sélectionné. À l'intérieur de cette fonction, il est possible de mettre en place une structure `Switch/case` par exemple qui va gérer les différents cas d'usage.

```
tabBuilder: (context, index) {
 switch (index) {
   case 0:
     return CupertinoTabView(builder: (context) {
       return CupertinoPageScaffold(
           child: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             Text('$index: Accueil'),
           ],
         ),
       ));
     });
   case 1:
     return CupertinoTabView(builder: (context) {
       return CupertinoPageScaffold(
           child: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             Text('$index: Mon compte'),
           ],
         ),
       ));
     });
   case 2:
     return CupertinoTabView(builder: (context) {
       return CupertinoPageScaffold(
           child: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             Text('$index: Statistiques'),
           ],
```

```
            ),
        ));
    });
  }
},
```

Il est intéressant de remarquer l'absence de `setState()`. Le même effet est réalisé grâce au `tabBuilder`. Chaque cas prévoit la construction d'un widget `CupertinoTabView` qui renvoie lui-même un nouveau `CupertinoPageScaffold`.

Voici le rendu :

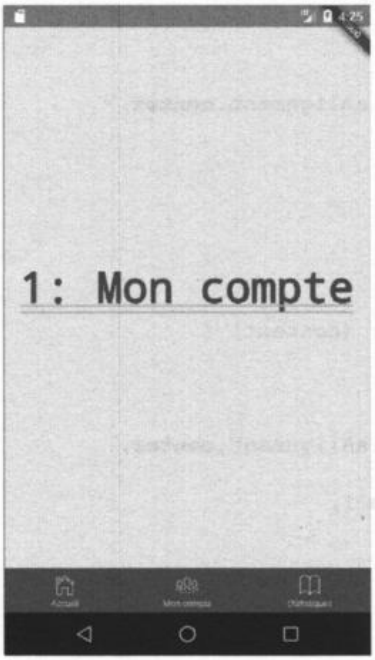


Figure 7 - CupertinoTabScaffold

© Editions ENI - All rights reserved

3.3 CupertinoNavigationBar

Sous iOS, il existe également une barre de navigation qui vient se placer en haut de l'écran, il s'agit de la `CupertinoNavigationBar`. Elle s'intègre parfaitement au sein du widget `CupertinoPageScaffold`.

La particularité est qu'elle peut comporter plusieurs widgets, généralement des icônes ou du texte, qui auront une position précise grâce à trois propriétés : `leading`, `middle` et `trailing`.

Voici un exemple de code d'un `Scaffold` au style iOS intégrant uniquement un texte :

```
@override
Widget build(BuildContext context) {
 return CupertinoPageScaffold(
   child: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'CupertinoNavigationBar',
           style: TextStyle(fontSize: 20, decoration: TextDecoration.none),
         )
       ],
     ),
   ),
  );
}
```

L'objectif est maintenant de rajouter la barre de navigation supérieure. Pour cela, ce `Scaffold` possède une propriété intitulée `navigationBar`. Il suffit alors d'y associer la `CupertinoNavigationBar`. Dans l'exemple les trois emplacements (`leading`, `middle`, `trailing`) vont être implémentés.

```
navigationBar: CupertinoNavigationBar(
 leading: Icon(
   Icons.arrow_back_ios,
   color: Colors.white,
 ),
 automaticallyImplyLeading: true,
 middle: Text(
   'Titre de la page',
   style: TextStyle(color: Colors.white),
 ),
```

```
    trailing: Icon(
      Icons.add_shopping_cart,
      color: Colors.white,
    ),
    backgroundColor: Colors.red,
  ),
```

Le résultat est le suivant :

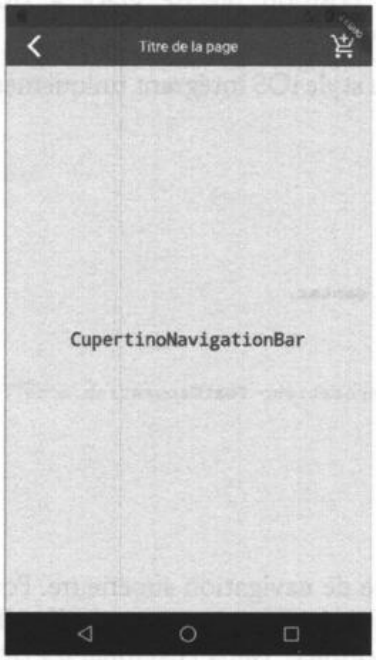


Figure 8 - CupertinoNavigationBar

Par défaut, dans le cadre d'une navigation, la propriété `middle` est un texte correspondant au titre de la page. La propriété `leading` correspond à un chevron permettant un retour arrière.

© Editions ENI - All rights reserved

4. Détection de la plateforme

Si l'objectif d'une application est de se fondre dans le style de son système d'exploitation hôte, il convient alors de lui faire posséder les deux styles : Material Design et Cupertino. Dès lors, un outil de détection de la plateforme est nécessaire pour savoir quel style sera chargé.

Pour tester cela, il faut, au préalable, créer deux Widgets, un pour Android et l'autre pour iOS. En l'occurrence, il s'agira respectivement d'un `Scaffold` et d'un `CupertinoPageScaffold`.

À l'intérieur de la classe `_MyHomePageState`, le code du widget Material Design correspondra à ceci :

```
Widget material() {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.red,
   ),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'style Material Design',
         ),
       ],
     ),
   ),
   bottomNavigationBar: BottomNavigationBar(
     items: const <BottomNavigationBarItem>[
       BottomNavigationBarItem(
           icon: Icon(
             Icons.home,
             color: Colors.white,
           ),
           title: Text(
             'Accueil',
             style: TextStyle(color: Colors.white),
           )),
       BottomNavigationBarItem(
           icon: Icon(
```

```
            Icons.supervised_user_circle,
            color: Colors.white,
          ),
          title: Text(
            'Mon compte',
            style: TextStyle(color: Colors.white),
          )),
        BottomNavigationBarItem(
          icon: Icon(
            Icons.assessment,
            color: Colors.white,
          ),
          title: Text(
            'Statistiques',
            style: TextStyle(color: Colors.white),
          )),
      ],
      backgroundColor: Colors.red,
    ),
  );
}
```

Cela reprend un écran déjà vu, composé d'une appBar et d'une bottomNavigationBar. Au centre, un texte va indiquer que le style est le Material Design.

Pour iOS, la même chose est réalisée. Cette fois-ci, le Widget se nomme cupertino :

```
Widget cupertino() {
  return CupertinoPageScaffold(
    child: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'style Cupertino',
            style: TextStyle(fontSize: 20, decoration: TextDecoration.none),
          )
        ],
      ),
    ),
    navigationBar: CupertinoNavigationBar(
      leading: Icon(
        Icons.arrow_back_ios,
        color: Colors.white,
```

© Editions ENI - All rights reserved

```
      ),
      automaticallyImplyLeading: true,
      middle: Text(
        widget.title,
        style: TextStyle(color: Colors.white),
      ),
      trailing: Icon(
        Icons.add_shopping_cart,
        color: Colors.white,
      ),
      backgroundColor: Colors.red,
    ),
  );
}
```

Sur iOS, un `CupertinoPageScaffold` est utilisé. À l'intérieur, une `navigationBar` est intégrée. Au centre, un texte confirme le style : « *style Cupertino* ».

Maintenant, il reste à mettre en place la détection de la plateforme. Pour cela, il convient de réaliser une importation :

```
import 'dart:io';
```

Cet import donne accès à une fonctionnalité primordiale : `Platform`. La classe `Platform` possède plusieurs méthodes renvoyant des booléens :

- `Platform.isAndroid` : pour détecter si le programme s'exécute sur Android.
- `Platform.isFuchsia` : pour détecter si le programme s'exécute sur Fuchsia.
- `Platform.isIOS` : pour détecter si le programme s'exécute sur iOS.
- `Platform.isLinux` : pour détecter si le programme s'exécute sur Linux.
- `Platform.isMacOS` : pour détecter si le programme s'exécute sur macOS.
- `Platform.isWindows` : pour détecter si le programme s'exécute sur Windows.

Dans l'exemple commencé précédemment, la détection va se faire dans le `Widget build`. Le retour va être doté d'un opérateur ternaire qui en fonction de la plateforme chargera le `Widget material` ou le `Widget cupertino` créés juste avant.

```
@override
Widget build(BuildContext context) {
 return Platform.isIOS ? cupertino() : material();
}
```

Voici le code complet de l'application :

```
import 'package:flutter/material.dart';
import 'package:flutter/cupertino.dart';
import 'dart:io';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Détection de la plateforme'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Platform.isIOS ? cupertino() : material();
 }

 Widget material() {
```

© Editions ENI - All rights reserved

```
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
        backgroundColor: Colors.red,
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'style Material Design',
            ),
          ],
        ),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
              icon: Icon(
                Icons.home,
                color: Colors.white,
              ),
              title: Text(
                'Accueil',
                style: TextStyle(color: Colors.white),
              )),
          BottomNavigationBarItem(
              icon: Icon(
                Icons.supervised_user_circle,
                color: Colors.white,
              ),
              title: Text(
                'Mon compte',
                style: TextStyle(color: Colors.white),
              )),
          BottomNavigationBarItem(
              icon: Icon(
                Icons.assessment,
                color: Colors.white,
              ),
              title: Text(
                'Statistiques',
                style: TextStyle(color: Colors.white),
              )),
        ],
        backgroundColor: Colors.red,
      ),
    );
  }

  Widget cupertino() {
```

```
    return CupertinoPageScaffold(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'style Cupertino',
              style: TextStyle(fontSize: 20, decoration: TextDecoration.none),
            )
          ],
        ),
      ),
      navigationBar: CupertinoNavigationBar(
        leading: Icon(
          Icons.arrow_back_ios,
          color: Colors.white,
        ),
        automaticallyImplyLeading: true,
        middle: Text(
          widget.title,
          style: TextStyle(color: Colors.white),
        ),
        trailing: Icon(
          Icons.add_shopping_cart,
          color: Colors.white,
        ),
        backgroundColor: Colors.red,
      ),
    );
  }
}
```

Pour vérifier si cela fonctionne correctement, il convient de procéder à un test. L'application va donc être lancée sur deux émulateurs différents, le Pixel 2 tournant sous Android et l'iPhone 11 Pro Max sur iOS.

© Editions ENI - All rights reserved

Sur le Pixel 2 le rendu est le suivant :

Figure 9 - Rendu Android avec un style Material Design

Sur iPhone 11 Pro Max, le rendu est celui-ci :

Figure 10 - Rendu iOS avec un style Cupertino

© Editions ENI - All rights reserved

5. Boutons

Dans le développement d'une application, les boutons sont essentiels. Ils garantissent la possibilité d'action à l'utilisateur. Les boutons peuvent prendre diverses formes. Cette partie va rendre compte des widgets de type bouton qui sont couramment utilisés avec Flutter.

5.1 FloatingActionButton

Le bouton d'action flottant, parfois appelé par son acronyme FAB, a été détaillé précédemment dans les sections AppBar et FloatingActionButton. Ce widget peut être considéré comme faisant partie du `Scaffold` contrairement aux autres boutons qui, eux, s'intègrent dans la propriété body du `Scaffold`, au sein d'autres widgets le plus souvent.

5.2 RaisedButton

Le `raisedButton` est un bouton Material Design. Il s'agit d'un bouton simulant un relief. Cet effet est donné par une surélévation. Cette dernière augmente davantage lors de l'appui. Sa taille minimum est de 88.0 par 36.0.

Côté action, il propose deux appels : `onPressed` et `onLongPressed`. S'ils sont nuls, le bouton est désactivé (*disable*) par défaut.

Remarque

Le changement de couleur du bouton n'est apparent que dans le cas où il est actif. Il faut donc alimenter une valeur non nulle pour `onPressed` et/ou `onLongPressed`.

Afin de voir l'aspect d'un `RaisedButton`, une application est créée avec simplement un `floatingActionButton` ainsi qu'un texte central qui va détecter un appui sur ce dernier :

```
class _MyHomePageState extends State<MyHomePage> {
 String _choixBtn = 'Aucun bouton';

 void _quelBouton(String btn) {
   setState(() {
     switch (btn) {
```

```
        case 'fab':
          {
            _choixBtn = "FloatingActionButton";
          }
        break;
      }
    });
  }
  void _fabOnPressed() {
    _quelBouton('fab');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Le bouton appuyé est : ',
            ),
            Text(
              '$_choixBtn',
              style: TextStyle(
                color: Colors.red, fontSize: 30, fontWeight: FontWeight.bold),
              ),
            ],
          ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _fabOnPressed,
        tooltip: 'Increment',
        child: Icon(Icons.add),
        backgroundColor: Colors.red,
      ),
    );
  }
}
```

© Editions ENI - All rights reserved

L'appui sur le floatingActionButton appelle la méthode _fabOnPressed. Cette dernière n'est qu'un relai vers une autre méthode, _quelBouton, qui prend en paramètre une chaîne de caractères correspondant au bouton qui vient d'être actionné. Dedans, une structure switch/case va prévoir l'affectation d'une valeur chaîne de caractères à la variable _choixBtn.

Maintenant, le but va être de rajouter un widget RaisedButton. Le code est inséré juste après celui du dernier widget Text :

```
Padding(padding: EdgeInsets.all(30)),
RaisedButton(
 onPressed: _raisedButtonOnPressed,
 child: Text('RaisedButton', style: TextStyle(color: Colors.white),),
 color: Colors.red,
 elevation: 8,
),
```

Le bouton comporte un texte de référence. La couleur rouge lui est attribuée. L'élévation est augmentée un peu pour apprécier son effet : sa valeur est passée à 8.

La détection de l'appui se fait sur onPressed évidemment. Comme pour son homologue, une méthode (_raisedButtonOnPressed) est créée pour gérer cet appui.

Celle-ci prévoit l'appel à la méthode _quelBouton :

```
void _raisedButtonOnPressed() {
 _quelBouton('raised');
}
```

Il convient maintenant de rajouter ce cas dans la structure switch/case de _quelBouton :

```
void _quelBouton(String btn) {
 setState(() {
   switch (btn) {
     case 'fab':
       {
         _choixBtn = "FloatingActionButton";
       }
       break;
     case 'raised':
```

```
        {
          _choixBtn = 'RaisedButton';
        }
        break;
      }
    });
  }
```

Cette application va permettre de détecter le bouton actionné et d'indiquer sur l'écran celui-ci, avec le rendu suivant :

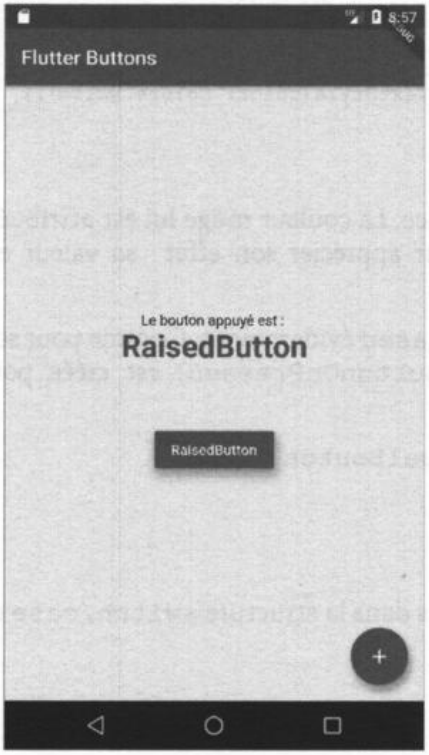


Figure 11 - RaisedButton

© Editions ENI - All rights reserved

5.3 FlatButton

Le `FlatButton` est également un bouton de style Material Design. Il possède des caractéristiques très proches de celles du `RaisedButton`. La différence notable se situe dans l'absence d'élévation. Il a donc un usage plus adapté aux fenêtres de type popup qui elle-même sont déjà surélevée par rapport à l'écran principal.

Pour rester simple au sujet des différentes propriétés, il existe bien évidemment au minimun `child` et `onPressed` (ainsi que `onLongPress`). Si une valeur nulle y est affectée, alors le bouton est désactivé par défaut.

Pour l'aspect graphique, il est offert la possibilité de jouer sur la couleur (`color`) du bouton ou celle du texte (`textColor`).

En reprenant l'exemple commencé plus haut, il suffit d'ajouter le code du `FlatButton` juste après celui du `raisedButton` :

```
FlatButton(
   onPressed: _flatButtonOnPressed,
   child: Text('FlatButton',),
   color: Colors.green,
   textColor: Colors.white,
   shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(30.0)),
),
```

Le bouton est coloré en vert pour le distinguer de l'autre. Pour obtenir un visuel un peu différent, les bords du bouton sont arrondis grâce à la propriété `shape`.

La propriété `onPressed` appelle `_flatButtonOnPressed` :

```
void _flatButtonOnPressed() {
 _quelBouton('flat');
}
```

Ici, il ne s'agit, une fois de plus, que d'un relais vers un cas prévu dans _quelBouton :

```
void _quelBouton(String btn) {
  setState(() {
    switch (btn) {
      case 'fab':
        {
          _choixBtn = "FloatingActionButton";
        }
        break;
      case 'raised':
        {
          _choixBtn = 'RaisedButton';
        }
        break;
      case 'flat':
        {
          _choixBtn = 'FlatButton';
        }
        break;
    }
  });
}
```

© Editions ENI - All rights reserved

Finalement, l'écran ressemble à ceci :

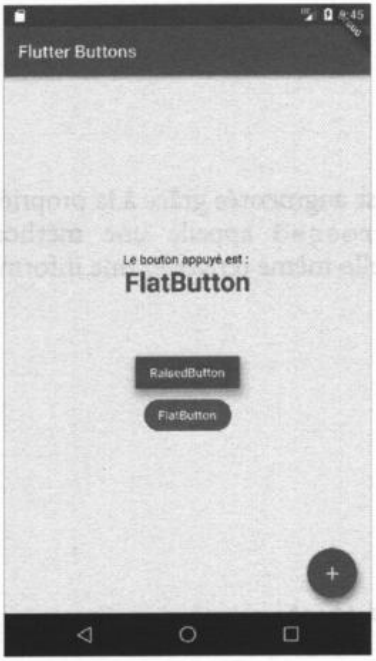


Figure 12 - FlatButton

5.4 IconButton

L'`IconButton` est, comme son nom le suggère, un bouton intégrant une icône. Il s'agit d'un bouton de style Material Design.

Dans la majorité des cas, ces boutons sont utilisés au sein d'`appBar`.

Pour alimenter ces `IconButton`, les icônes existantes sont nombreuses. La section Icones du présent chapitre y est consacrée et développe cet aspect.

Afin d'illustrer ces propos, un `IconButton` est ajouté à la suite du code :

```
IconButton(
 icon: Icon(Icons.accessibility),
 onPressed: _iconButtonOnPressed,
 color: Colors.blue,
 iconSize: 40,
),
```

L'icône choisie est colorée en bleu. Sa taille est augmentée grâce à la propriété `iconSize`. Comme précédemment, `onPressed` appelle une méthode dédiée (`_iconButtonOnPressed`) qui va elle-même renvoyer une information à la méthode `_quelBouton`.

```
void _iconButtonOnPressed() {
 _quelBouton('icon');
}
```

```
void _quelBouton(String btn) {
 setState(() {
   switch (btn) {
     case 'fab':
       {
         _choixBtn = "FloatingActionButton";
       }
       break;
     case 'raised':
       {
         _choixBtn = 'RaisedButton';
       }
       break;
     case 'flat':
       {
         _choixBtn = 'FlatButton';
       }
       break;
     case 'icon':
       {
         _choixBtn = 'IconButton';
       }
       break;
       }
 });
}
```

© Editions ENI - All rights reserved

Avec l'ajout de ce bouton, l'écran complet s'affiche dorénavant comme ceci :

Figure 13 - IconButton

Ce bouton ne comporte pas de fond coloré. Il n'existe pas de propriété dans la classe `IconButton` qui permette d'en définir. Cependant, il y a une autre possibilité qui consiste à créer un widget `Ink` permettant la coloration d'un environnement. Il suffit ensuite de rajouter l'`IconButton` à l'intérieur de celui-ci, et le résultat obtenu est très satisfaisant.

Dans l'exemple, un nouveau widget `Ink` est mis en place juste après l'`IconButton` :

```
Ink(
 decoration: const ShapeDecoration(
   shape: CircleBorder(),
   color: Colors.blueAccent,
 ),
 child: IconButton(
```

```
    icon: Icon(Icons.accessibility_new),
    onPressed: _iconButtonCircleOnPressed,
    color: Colors.white,
  ),
)
```

Le widget `Ink` possède une propriété `decoration` qui peut intégrer une couleur de fond (`color`) ainsi qu'une forme (`shape`). Dans le cas présent, la couleur porte sur un bleu légèrement plus soutenu que sur l'icône d'avant et la forme est arrondie (`CircleBorder`).

En guise de `child`, un `IconButton` est inséré. Afin de le distinguer de son collègue, une nouvelle méthode `_iconButtonCircleOnPressed` est créée :

```
void _iconButtonCircleOnPressed() {
 _quelBouton('iconCircle');
}
```

La méthode `_quelBouton` est modifiée en conséquence :

```
void _quelBouton(String btn) {
 setState(() {
   switch (btn) {
     case 'fab':
       {
         _choixBtn = "FloatingActionButton";
       }
       break;
     case 'raised':
       {
         _choixBtn = 'RaisedButton';
       }
       break;
     case 'flat':
       {
         _choixBtn = 'FlatButton';
       }
       break;
     case 'icon':
       {
         _choixBtn = 'IconButton';
       }
       break;
```

© Editions ENI - All rights reserved

```
        case 'iconCircle':
          {
            _choixBtn = 'IconButton avec fond';
          }
          break;
      }
    });
  }
```

Le rendu permet d'apprécier le fond coloré de cet `IconButton` :

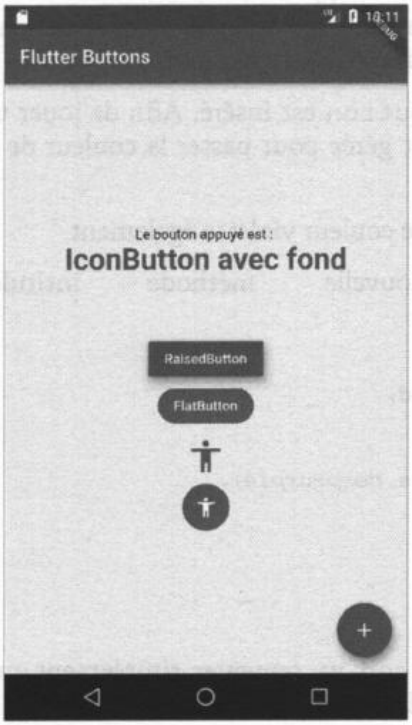


Figure 14 - IconButton coloré grâce au widget Ink

5.5 OutlineButton

L'OutlineButton est un bouton similaire au FlatButton. Il est de style Material Design. Pour être précis, il s'agit d'un bouton comportant une bordure externe. Pour configurer cette dernière, il existe une propriété nommée borderSide.

Par défaut, le bouton est entouré d'une bordure grise continue d'un pixel d'épaisseur. Le fond du bouton est transparent.

Comme pour les autres boutons énumérés avant, il existe des événements onPressed et onLongPressed qui le désactivent en cas de valeur nulle.

Pour alimenter l'exemple, un OutlineButton est inséré. Afin de jouer un peu avec les propriétés, borderSide est gérée pour passer la couleur de la bordure en violet.

L'objet child est simplement un texte de couleur violette également.

OnPressed appelle une nouvelle méthode intitulée _outlineButtonOnPressed.

```
OutlineButton(
 onPressed: _outlineButtonOnPressed,
 child: Text(
     'OutlineButton',
     style: TextStyle(color: Colors.deepPurple),
 ),
 borderSide: BorderSide(
   color: Colors.deepPurple,
 ),
),
```

La méthode _outlineButtonOnPressed va renvoyer simplement une information vers _quelBouton à laquelle est ajouté ce dernier cas.

```
void _outlineButtonOnPressed() {
 _quelBouton('outline');
}
```

© Editions ENI - All rights reserved

```
void _quelBouton(String btn) {
 setState(() {
   switch (btn) {
     case 'fab':
       {
         _choixBtn = "FloatingActionButton";
       }
       break;
     case 'raised':
       {
         _choixBtn = 'RaisedButton';
       }
       break;
     case 'flat':
       {
         _choixBtn = 'FlatButton';
       }
       break;
     case 'icon':
       {
         _choixBtn = 'IconButton';
       }
       break;
     case 'iconCircle':
       {
         _choixBtn = 'IconButton avec fond';
       }
       break;
     case 'outline':
       {
         _choixBtn = 'OutlineButton';
       }
       break;
   }
 });
}
```

Avec l'ajout de ce nouveau bouton, le rendu évolue et ressemble maintenant à ceci :

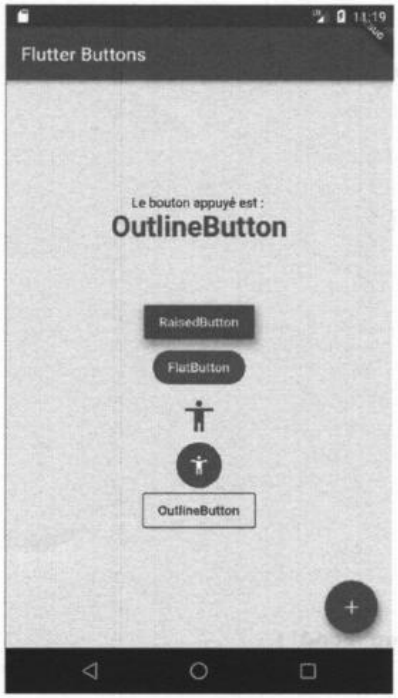


Figure 15 - OutlineButton

5.6 CupertinoButton

Comme son nom le sous-entend, il s'agit d'un bouton au style iOS. Il est possible d'y inclure indifféremment un texte ou une icône en tant que `child`. Plusieurs propriétés sont accessibles. `BorderRadius` permet de gérer l'arrondi du bouton (par défaut, il est à 8), `color` offre la possibilité de colorer le bouton avec la couleur de son choix.

Comme toujours, un événement `onPressed` permet de définir les actions qui auront lieu lors de l'appui.

© Editions ENI - All rights reserved

Pour finir avec l'exemple et mettre en lumière le CupertinoButton, celui-ci va être ajouté au code :

```
CupertinoButton(
   child: Text('CupertinoButton'),
   onPressed: _cupertinoOnPressed,
   color: Colors.teal,
)
```

L'unique enfant associé est un texte. Pour varier, le bouton est colorié avec du bleu turquoise.

Dans le cas présent, et pour conserver la nomenclature employée, onPressed appelle la méthode _cupertinoButtonOnPressed. Voici le code qui s'y trouve :

```
void _cupertinoOnPressed() {
 _quelBouton('cupertino');
}
```

Le CupertinoButton est le dernier cas de _quelBouton :

```
void _quelBouton(String btn) {
 setState(() {
   switch (btn) {
     case 'fab':
       {
         _choixBtn = "FloatingActionButton";
       }
       break;
     case 'raised':
       {
         _choixBtn = 'RaisedButton';
       }
       break;
     case 'flat':
       {
         _choixBtn = 'FlatButton';
       }
       break;
     case 'icon':
       {
         _choixBtn = 'IconButton';
       }
       break;
     case 'iconCircle':
       {
```

```
        _choixBtn = 'IconButton avec fond';
      }
      break;
    case 'outline':
      {
        _choixBtn = 'OutlineButton';
      }
      break;
    case 'cupertino':
      {
        _choixBtn = 'CupertinoButton';
      }
      break;
  }
});
}
```

Il est intéressant de voir sur un seul écran l'ensemble des boutons existants à ce jour au sein de Flutter. Voici donc le rendu final incluant le `CupertinoButton` :

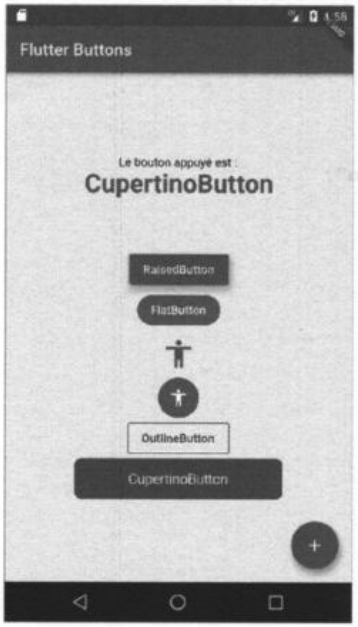


Figure 16 - CupertinoButton

© Editions ENI - All rights reserved

6. Médias

Dans le développement d'applications, il est indispensable de rendre l'expérience utilisateur la plus réussie possible. Il est donc primordial de savoir intégrer des médias. Dans cette section, le sujet va porter sur les icônes et les images.

6.1 Icônes

Les icônes sont l'un des outils les plus employés de nos jours. Elles permettent d'obtenir une compréhension rapide simplement au travers d'un visuel. De plus, à force de les employer, des conventions se sont mises en place. Par exemple, tout le monde associe la loupe à une recherche ou une roue dentée aux paramètres.

L'insertion de l'icône peut être réalisée au travers du widget `Icon`. Ce dernier permet de jouer avec certaines propriétés. Il est notamment possible de définir la taille de l'icône grâce à `size`. Sa couleur peut également être modifiée via `color`.

En dehors de ces paramètres optionnels, la classe `Icon` attend obligatoirement un objet de type `IconData`. C'est le cas pour les `Icons` et les `CupertinoIcons`.

Flutter offre un vaste choix d'icônes. Faciles à trouver, puisqu'il existe un choix important déjà prévu. Il propose deux possibilités de base, soit en travaillant avec la classe `Icons` qui comporte une collection au style Material Design, soit avec la classe `CupertinoIcons` qui propose un style iOS.

6.1.1 Icons (Material Design)

Il est possible de retrouver l'ensemble de ces icônes de cette classe à cet endroit : https://material.io/resources/icons/?style=baseline

Voici la mise en place d'une application simple comportant une icône de style Material Design (`Icons`) :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Icônes Flutter'),
   );
 }
}
class MyHomePage extends StatelessWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Row(
             mainAxisAlignment: MainAxisAlignment.center,
             children: <Widget>[
```

© Editions ENI - All rights reserved

```
                Text(
                  'Icône Material Design :',
                  style: TextStyle(
                      color: Colors.teal,
                      fontSize: 25,
                      fontWeight: FontWeight.bold
                  ),
                ),
                Padding(padding: EdgeInsets.all(10)),
                Icon(
                  Icons.home,
                  color: Colors.green,
                  size: 35,
                ),
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

Dans ce code, un texte et une icône sont mis en place. Cette dernière représente une maison. Ces deux éléments sont insérés dans un widget `Row` qui sert à créer une ligne. De plus, afin de créer un espacement minimum entre eux, un autre widget, `Padding`, est utilisé. Ces sujets seront abordés un peu plus loin dans ce chapitre.

Pour l'instant, le rendu est simple :

Figure 17 - Icône Material Design

6.1.2 CupertinoIcons (Style iOS)

Il est possible de retrouver l'ensemble des icônes de cette classe à cet endroit : https://github.com/flutter/cupertino_icons/blob/master/map.png

Dans l'exemple précédent, une nouvelle ligne (Row) est créée avec un texte et une icône représentant à nouveau une maison, mais cette fois-ci au style iOS. Ce code est donc ajouté :

```
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
   Text(
     'Icône iOS :',
     style: TextStyle(
```

© Editions ENI - All rights reserved

```
            color: Colors.blueGrey,
            fontSize: 25,
            fontWeight: FontWeight.bold
        ),
    ),
    Padding(padding: EdgeInsets.all(10)),
    Icon(
      CupertinoIcons.home,
      color: Colors.black,
      size: 35,
    ),
  ],
),
```

Les dimensions sont conservées mais les couleurs sont modifiées pour mieux identifier les différents éléments de l'écran. Celui-ci prend dorénavant l'aspect suivant :

Figure 18 - CupertinoIcons

6.1.3 FontAwesome

FontAwesome est un site Internet bien connu des développeurs web. Il offre une collection très importante d'icônes en tout genre. Pour le découvrir, il est accessible via l'URL : https://fontawesome.com

À ce jour, il référence plus de 7700 icônes dont environ 1500 gratuites.

Par défaut, elles ne sont pas accessibles directement dans Flutter mais il existe un moyen d'y avoir accès.

L'équipe de développement de Flutter, ainsi que sa communauté, participe à la création de packages. Ils sont centralisés sur le site https://pub.dev.

En développement, le fait d'ajouter des packages ou des bibliothèques à un projet s'appelle la gestion de dépendances. Flutter en assure l'organisation au sein d'un fichier précis : *pubspec.yaml*.

Concernant FontAwesome, un package permet d'y avoir accès. Il se nomme **font_awesome_flutter**. À ce jour, le site pub.dev le propose dans sa version 8.5.0. :

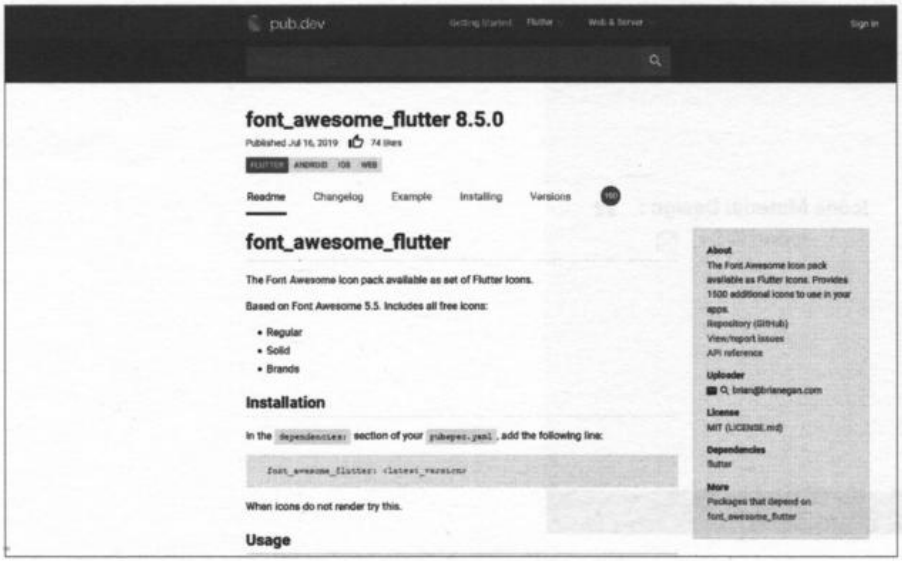


Figure 19 - Package font_awesome_flutter 8.5.0 sur le site pub.dev

© Editions ENI - All rights reserved

Le site donne toutes les explications nécessaires quant à l'utilisation du package. Au-delà, il permet aussi d'obtenir un descriptif, de connaître les auteurs de ce dernier ainsi que la licence d'utilisation.

Pour obtenir ce package, il faut se placer dans le fichier *pubspec.yaml*. Sous `dependencies`, il suffit de rajouter la ligne suivante :

```
font_awesome_flutter: ^8.5.0
```

Dans le projet commencé ci-dessus, les dépendances du *pubspec.yaml* devraient être celles-ci :

```
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
  font_awesome_flutter: ^8.5.0
```

Attention, ce code est valable au moment de la rédaction de l'ouvrage. Il convient de toujours vérifier que la version visée est la même et qu'elle n'a pas évolué. Si tel était le cas, il suffirait de modifier le numéro de version pour le faire correspondre à celle du package trouvé sur pub.dev.

Il reste une étape cruciale. Maintenant que la dépendance est déclarée, il faut en récupérer les données. Pour cela, il faudra cliquer sur **Packages get** en haut à droite de l'écran dans Android Studio. Attention, cette option n'est disponible que sur le fichier *pubspec.yaml*.

Figure 20 - Récupération des packages grâce à Packages get

Dans la console, le traitement devrait s'opérer. Si tout s'est bien déroulé, le code de sortie obtenu est 0 :

```
C:\src\flutter\flutter\bin\flutter.bat --no-color packages get
Running "flutter pub get" in flutter_widget6...                    1,5s
Process finished with exit code 0
```

Figure 21 - Résultat de Packages get

À partir de cet instant, ce package est disponible et peut être utilisé dans le code du programme. Dans l'exemple commencé un peu plus haut, une dernière ligne (Row) est insérée :

```
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
   Text(
     'Icône font_awesome :',
     style: TextStyle(
         color: Colors.deepPurple,
         fontSize: 25,
         fontWeight: FontWeight.bold
     ),
   ),
   Padding(padding: EdgeInsets.all(10)),
   Icon(
     FontAwesomeIcons.home,
     color: Colors.purple,
     size: 35,
   ),
 ],
),
```

Une nouvelle classe est dorénavant accessible. Il s'agit de `FontAwesomeIcons`. Celle-ci a été obtenue grâce à la récupération du package effectuée juste avant. Pour rester cohérent, une icône en forme de maison est encore affectée. Les couleurs sont modifiées. Le reste ne bouge pas.

© Editions ENI - All rights reserved

Avant d'examiner le rendu, il est intéressant de regarder les importations :

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:font_awesome_flutter/font_awesome_flutter.dart';
```

L'utilisation des `CupertinoIcons` et du package `font_awesome_flutter` est apparente ici. En effet, c'est bien grâce aux importations que l'utilisation des classes `CupertinoIcons` et `FontAwesomeIcons` est possible.

Voyons maintenant à quoi ressemble notre écran après l'ajout de cette dernière icône :

Figure 22 - FontAwesomeIcons

6.2 Images

Pour rendre une application agréable en termes d'expérience utilisateur, il est courant d'insérer des images. Avec Flutter, il existe aussi un widget dédié : `Image`.

Pour cela, plusieurs constructeurs nommés sont mis à disposition. Ils correspondent aux différents types de provenance d'une image, celles incluses au projet ou celles venant directement du web.

Les formats d'images pris en compte sont nombreux : JPEG, PNG, GIF, WebP, AnimatedWebP, BMP et WBMP.

6.2.1 Images du projet

Pour insérer une image directement dans le projet, il suffit de créer un dossier à sa racine. Dans l'exemple suivant, un premier dossier nommé Assets est créé, puis un sous-dossier nommé Images. Pour créer un dossier, il suffit de faire un clic droit sur le nom du projet puis d'aller sur **New** et enfin sur **Directory**.

Figure 23 - Création de dossiers pour les images

© Editions ENI - All rights reserved

Pour cet exemple, le logo de Flutter va être utilisé. Le fichier s'appelle *flutter_logo_tm.png*. Pour l'insérer, il suffit de copier/coller le fichier directement dans le sous-dossier *Images* qui vient d'être ajouté à l'application. La fenêtre suivante devrait s'ouvrir :

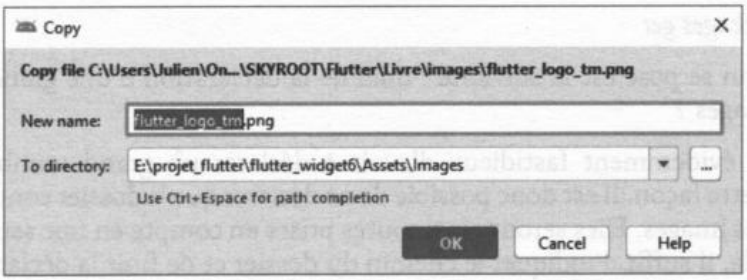


Figure 24 - Insertion d'un fichier

Dans cette fenêtre, il est possible de renommer le fichier. C'est également l'occasion de vérifier l'endroit où il va être inséré. Pour valider l'insertion, il suffit de cliquer sur **OK**.

Maintenant, pour pouvoir utiliser les images, il va falloir retourner dans le fichier *pubspec.yaml*. À l'intérieur, dans la section « `flutter :` » il faudra décommenter l'exemple d'insertion d'assets :

```
# assets:
#  - images/a_dot_burr.jpeg
#  - images/a_dot_ham.jpeg
```

Attention, il s'agit d'un fichier yaml. Il s'agit donc d'un fichier de configuration où l'indentation a une importance cruciale, il faut absolument veiller à la respecter ; `assets`, par exemple, doit être inscrit à deux espaces de la marge gauche.

Pour déclarer l'image dans les assets, il faudra indiquer son chemin d'accès. Voici, ce qu'il faut écrire :

```
flutter:
 uses-material-design: true

 assets:
   - Assets/Images/flutter_logo_tm.png
```

Une fois cette déclaration effectuée, il convient de nouveau de cliquer sur **Packages get** afin qu'elle soit prise en compte.

Figure 24 - Packages get

La question qui se pose est la suivante : quid de la déclaration d'une grande quantité d'images ?

Il serait bien évidemment fastidieux d'avoir à déclarer un grand nombre d'images de cette façon. Il est donc possible de ne déclarer que le dossier contenant toutes les images. Elles seront ainsi toutes prises en compte en une seule ligne. Pour cela, il suffit d'indiquer le chemin du dossier et de finir la déclaration par « / ». Pour l'exemple ci-dessus, cela donnerait :

```
flutter:
  uses-material-design: true
 assets:
   - Assets/Images/
```

Maintenant que la déclaration est effective, il ne reste plus qu'à intégrer cette image dans le projet. Le widget `Image` possède un constructeur nommé pour les assets : `Image.asset`. Il attend a minima le chemin de l'image.

```
Image.asset('Assets/Images/flutter_logo_tm.png'),
```

L'image rendue est un peu trop grande. Pour ajuster sa taille, il est possible d'ajuster la propriété `width`.

```
Image.asset(
   'Assets/Images/flutter_logo_tm.png',
   width: 150,
),
```

© Editions ENI - All rights reserved

Grâce à cette opération, l'image est moins grande. Le rendu de l'application a donc évolué comme ceci :

Figure 25 - Insertion d'une image (assets)

6.2.2 Images du Web

Si l'insertion des images directement dans le projet est une bonne façon de maitriser ses ressources, il est aussi possible de récupérer une image depuis le Web directement. Cette technique est évidemment plus simple et moins coûteuse en stockage. Mais elle présente un risque non négligeable, celui de voir la ressource disparaître et devenir inaccessible. C'est le cas, notamment, quand l'application n'arrive plus à se connecter au réseau.

Pour inclure des images provenant du web, le widget Image propose un constructeur nommé Image.network qui attend a minima une URL.

Pour faire suite au code déjà présenté, cette ligne est rajoutée :

```
Image.network('https://www.editions-eni.fr/styles/france/
images/logo_ENI.png'),
```

L'enjeu est de récupérer le logo de ENI Éditions présent sur le site web à l'adresse : https://www.editions-eni.fr/styles/france/images/logo_ENI.png

Dans l'émulateur, l'opération fonctionne correctement. L'affichage rend ceci :

Figure 26 - Insertion d'une image provenant du Web

© Editions ENI - All rights reserved

6.3 Texte

6.3.1 Text

Au cours des divers exemples étudiés, les textes sont apparus régulièrement sans qu'ils soient vraiment expliqués. Leur insertion est réalisée par le biais du widget `Text`.

Par défaut, il faut au moins renseigner une chaîne de caractères. Il s'agit du seul paramètre obligatoire du constructeur. Toutefois, il existe d'autres paramètres qui permettent de travailler autour de ce texte. C'est le cas de `style` qui offre la possibilité d'arranger son aspect. L'alignement est modulable avec la propriété `textAlign`. Quant à `textOverflow`, elle permet de définir la conduite à tenir si le texte est trop grand et occupe plus d'une ligne.

Voici un exemple :

```
Text(
 'Voici un texte quelconque',
 style: TextStyle(
   color: Colors.brown,
   fontWeight: FontWeight.bold,
   fontSize: 50,
   fontStyle: FontStyle.italic
 ),
 textAlign: TextAlign.left,
),
```

Pour ce widget, la phrase « Voici un texte quelconque » est inscrite.

6.3.2 Mise en forme du texte

Pour agrémenter la chose, la propriété `style` reçoit un widget `TextStyle`. Il s'agit d'un widget qui, comme son nom le laisse entendre, permet de mettre en forme un texte. Il est possible de travailler la couleur (`color`), le poids de la police (`fontWeight`), sa taille (`fontSize`) ou encore son style (`fontStyle`).

La propriété `textAlign` permet de définir comment le texte va se positionner. À cet effet, il suffit de faire appel au widget `TextAlign` qui comporte différentes valeurs telles que `center`, `left`, `right`...

Le code ci-dessus est ajouté à l'application juste après celui des icônes et des images. L'écran prend maintenant cette forme :

Figure 27 - Mise en place d'un texte

Sur l'image ci-dessus, le texte est relativement grand et retourne à la ligne. L'utilisation de `overflow` permet de rectifier cela et d'imposer un comportement bien précis. Ici, le texte est placé directement dans la page (dans le widget `Colum` pour être précis) et n'a pas de contrainte en hauteur. Les effets de `overflow` sont donc limités. Il devient vraiment utile quand le texte s'intègre dans un élément restreint en hauteur et en largeur.

Pour montrer les différentes possibilités, le code ci-dessus est placé dans un widget `Container` (ce widget sera détaillé un peu plus loin dans ce chapitre) limité en largeur et en hauteur. Voici le code modifié :

```
Container(
 width: MediaQuery.of(context).size.width / 1.2,
 height: 100,
```

© Editions ENI - All rights reserved

```
    color: Colors.black,
    child: Text(
      'Voici un texte quelconque',
      style: TextStyle(
          color: Colors.brown,
          fontWeight: FontWeight.bold,
          fontSize: 50,
          fontStyle: FontStyle.italic
      ),
      textAlign: TextAlign.left,
      overflow: TextOverflow.clip
    ),
  ),
```

Le rendu évolue et le texte se présente dorénavant comme ceci :

Figure 28 - Texte placé dans un élément trop petit

Le widget `TextOverflow` comporte différentes valeurs utilisables dans ce cas :

- `Clip` : coupe le texte qui dépasse (valeur par défaut). Sur l'exemple, il n'y a donc aucun changement apparent.

Figure 29 - TextOverflow.clip

- `Ellipsis` : la partie du texte qui dépasse est remplacée par des points de suspension '...'.

Figure 30 - TextOverflow.ellipsis

- `Fade` : le texte qui dépasse est coupé par un fondu.

Figure 31 - TextOverflow.fade

- `Visible` : le texte est vu entièrement et dépasse de l'élément parent.

Figure 32 - TextOverflow.visible

© Editions ENI - All rights reserved

6.3.3 Choix de la police

Un autre paramètre important est la police choisie. Pour se différencier, les applications doivent souvent pousser la personnalisation le plus possible. Le choix d'une police confère une identité visuelle propre et indispensable à toute bonne application.

Au sein du widget `TextStyle`, le constructeur accepte un paramètre nommé `fontFamily` qui permet d'indiquer la volonté d'utiliser une autre police.

Mais avant cela, il convient de respecter quelques étapes préalables.
D'abord, il faut stocker une police dans le projet. Pour reprendre l'exemple de tout à l'heure, un second sous-dossier nommé Font est créé dans le dossier Assets. La police *Fruktur* de Viktoria Grabowska est téléchargée depuis Google Fonts et copiée dans le dossier nouvellement créé.

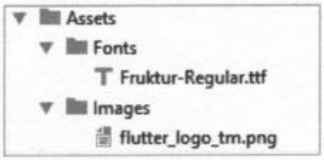


Figure 34 - Mise en place d'une police

L'étape suivante consiste à alimenter le fichier *pubspec.yaml* afin d'y déclarer cette police. De la même manière qu'il a été procédé pour les images, une ligne `fonts :` suivie de la déclaration de son nom (`- family`) utilisable postérieurement dans l'application et des chemins des fichiers (`- asset`) qui y sont associés doivent être insérés.

Le fichier *pubspec.yaml* de l'exemple s'enrichit et devient donc :

```
flutter:
 uses-material-design: true

 assets:
   - Assets/Images/

 fonts:
   - family: Fruktur
     fonts:
       - asset: Assets/Fonts/Fruktur-Regular.ttf
```

Pour cette nouvelle police, le nom Fruktur est indiqué :

```
- family: Fruktur
```

Ensuite le chemin de la police est énoncé et pointe vers le fichier copié juste avant dans le sous-dossier Fonts.

```
- asset: Assets/Fonts/Fruktur-Regular.ttf
```

Il ne reste plus qu'à cliquer de nouveau sur **Packages get** pour mettre à jour le projet.

Tout est en ordre pour pouvoir travailler avec cette nouvelle police. Pour vérifier son bon fonctionnement, deux textes sont insérés dans l'application. Le premier fera office de point de référence et de modèle. Le second se verra appliquer la nouvelle police. Ce code est ajouté à la suite :

```
Text(
 'Police par défaut',
 style: TextStyle(fontSize: 20),
),
Text(
 'Police Fruktur',
 style: TextStyle(
     fontSize: 20,
     color: Colors.purple,
     fontFamily: 'Fruktur'
 ),
),
```

Sur le second texte, la police Fruktur est assignée comme valeur à `fontFamily`. L'écran évolue et affiche :

Figure 35 - Texte avec police Fruktur

© Editions ENI - All rights reserved

6.3.4 Google Fonts

L'insertion d'une police autre que celle proposée par défaut a été décrite dans la section précédente. La procédure est assez lourde mais elle a l'avantage d'être relativement sécurisée puisque la police est embarquée dans l'application.

Il est toutefois possible, depuis peu, d'utiliser les polices de Google Fonts directement sans avoir à les intégrer au projet. L'ennui reste les problèmes relatifs à la connexion à Internet. En effet, en cas d'absence de connexion au réseau la police ne sera pas chargée. En revanche, l'application est plus légère puisqu'elle n'a pas à héberger les fichiers des différentes polices utilisées.

Cette fonctionnalité a été annoncée lors de la conférence *Flutter Interact* qui s'est déroulée le 11 décembre 2019 à Londres. Un package est disponible sur le site *pub.dev* sous le nom de *google_fonts 1.1.0*.

Pour retrouver le package, il suffit de suivre ce lien :
https://pub.dev/packages/google_fonts

Pour l'utiliser, il faut commencer par l'ajouter dans les dépendances du fichier *pubspec.yaml*.

```
Dependencies:
 flutter:
   sdk: flutter

 cupertino_icons: ^0.1.2
 font_awesome_flutter: ^8.5.0
 google_fonts: ^1.1.0
```

Une fois cette opération effectuée, il ne faut pas oublier de cliquer sur le lien **Packages get** afin que le package soit téléchargé et utilisable par la suite.

Dans le fichier *main.dart*, il convient maintenant de rajouter l'importation qui va permettre la manipulation de ce package :

```
import 'package:google_fonts/google_fonts.dart';
```

À partir de là, il suffit d'appeler le widget `GoogleFonts` sur un texte. Pour illustrer cela, le texte « Police Pacifico » est ajouté à l'exemple précédent. La propriété `style` appelle le widget `GoogleFonts` puis précise la police avec laquelle il faut travailler.

Il est possible de rajouter une personnalisation supplémentaire grâce à `TextStyle` pour modifier la couleur, la taille…

```
Text(
    'Police Pacifico',
    style: GoogleFonts.pacifico(
      textStyle: TextStyle(
        fontSize: 20,
        color: Colors.blue
      ),
    ),
),
```

Voici le rendu final de l'écran :

Figure 36 - Mise en place d'une Google Font

© Editions ENI - All rights reserved

6.4 Layouts (mise en page)

Afin de positionner les éléments convenablement, Flutter intègre un choix important de widgets. Ils peuvent prendre la forme de conteneurs, ou bien donner une orientation dans le placement des widgets enfants. Certains d'entre eux ont déjà été aperçus dans les différents exemples présentés mais sans jamais vraiment être expliqués.

6.4.1 Center

Center est un des premiers widgets découvert. Il fait partie du projet de démonstration et il est employé régulièrement. Comme son nom l'indique, il a vocation à centrer le widget qui sera son enfant que cela soit en hauteur comme en largeur.

Il est possible de jouer sur la dimension en utilisant `widthFactor` et `heightFactor`. Ils représentent des coefficients. Celui-ci s'applique sur l'élément enfant. Par exemple, si `heightFactor` est égal à 3, alors la hauteur de `Center` sera trois fois la hauteur du widget enfant qu'il contient.

Voici un exemple de code pour illustrer ce widget :

```
Center(
child: null,
heightFactor: 3);
```

6.4.2 Column

Le widget `Column` a été aussi beaucoup manipulé depuis le début. Il est intégré dans l'application de base avec le compteur d'incrémentation.

Sa particularité est de permettre d'aligner verticalement une liste de widgets enfants. En revanche, il ne permet pas de défilement sur cet axe (voir chapitre Listes et grilles, les `ListView`). Les enfants s'espacent dans la colonne selon la propriété `MainAxisAlignment`. Ses différentes valeurs possibles sont :

- `Start` : les éléments sont regroupés en haut de la colonne.
- `End` : les éléments sont regroupés en bas de la colonne.
- `Center` : les éléments sont regroupés au centre de la colonne.

- SpaceEvenly : les éléments sont répartis proportionnellement dans la colonne avec des marges équivalentes entre eux et avec la marge.
- SpaceAround : les éléments sont répartis proportionnellement dans la colonne avec des marges réduites aux extrémités.
- SpaceBetween : les éléments sont répartis proportionnellement dans la colonne sans marge aux extrémités.

Afin de mieux visualiser les différences entre ces positionnements et mettre en pratique les colonnes, cet exemple de code est écrit dans une nouvelle application :

```
Column(
 mainAxisAlignment: MainAxisAlignment.start,
 children: <Widget>[
   Icon(
     Icons.android,
     color: Colors.amber,
   ),
   Icon(
     Icons.home,
     color: Colors.amber,
   ),
   Icon(
     Icons.build,
     color: Colors.amber,
   ),
   Icon(
     Icons.phone,
     color: Colors.amber,
   ),
   Icon(
     Icons.group,
     color: Colors.amber,
   )
 ],
),
Column(
 mainAxisAlignment: MainAxisAlignment.end,
 children: <Widget>[
   Icon(
     Icons.android,
     color: Colors.deepOrange,
```

© Editions ENI - All rights reserved

```
    ),
    Icon(
      Icons.home,
      color: Colors.deepOrange,
    ),
    Icon(
      Icons.build,
      color: Colors.deepOrange,
    ),
    Icon(
      Icons.phone,
      color: Colors.deepOrange,
    ),
    Icon(
      Icons.group,
      color: Colors.deepOrange,
    )
  ],
),
Column(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
    Icon(
      Icons.android,
      color: Colors.blue,
    ),
    Icon(
      Icons.home,
      color: Colors.blue,
    ),
    Icon(
      Icons.build,
      color: Colors.blue,
    ),
    Icon(
      Icons.phone,
      color: Colors.blue,
    ),
    Icon(
      Icons.group,
      color: Colors.blue,
    )
  ],
),
```

```
Column(
 mainAxisAlignment: MainAxisAlignment.spaceEvenly,
 children: <Widget>[
   Icon(
     Icons.android,
     color: Colors.red,
   ),
   Icon(
     Icons.home,
     color: Colors.red,
   ),
   Icon(
     Icons.build,
     color: Colors.red,
   ),
   Icon(
     Icons.phone,
     color: Colors.red,
   ),
   Icon(
     Icons.group,
     color: Colors.red,
   )
 ],
),
Column(
 mainAxisAlignment: MainAxisAlignment.spaceAround,
 children: <Widget>[
   Icon(
     Icons.android,
     color: Colors.purple,
   ),
   Icon(
     Icons.home,
     color: Colors.purple,
   ),
   Icon(
     Icons.build,
     color: Colors.purple,
   ),
   Icon(
     Icons.phone,
     color: Colors.purple,
   ),
```

© Editions ENI - All rights reserved

```
    Icon(
      Icons.group,
      color: Colors.purple,
    )
  ],
),
Column(
 mainAxisAlignment: MainAxisAlignment.spaceBetween,
 children: <Widget>[
   Icon(
     Icons.android,
     color: Colors.green,
   ),
   Icon(
     Icons.home,
     color: Colors.green,
   ),
   Icon(
     Icons.build,
     color: Colors.green,
   ),
   Icon(
     Icons.phone,
     color: Colors.green,
   ),
   Icon(
     Icons.group,
     color: Colors.green,
   )
 ],
),
```

Dans ce code, il y a six colonnes qui possèdent chacune une valeur de `MainAxisAlignment` différente. Pour que cette différence soit visible, chaque colonne intègre les cinq mêmes icônes (le robot Android, une maison, une clé à molette, un téléphone et un groupe de personnes).

Pour que ce code soit complet, il manque encore certaines choses et notamment des Row qui seront nécessaires. C'est justement le sujet abordé dans la prochaine section. En attendant de les découvrir, voici malgré tout le rendu de l'application qui permet de vraiment visualiser les effets des divers alignements mis en place.

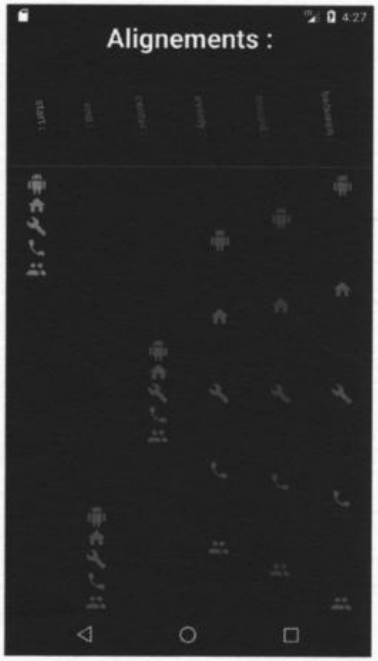


Figure 37 - Row et Column avec différents alignements

6.4.3 Row

Le widget `Row` est très semblable à `Column` à la différence que l'axe dans lequel viennent se placer les widgets enfants n'est plus vertical mais horizontal. Il permet d'intégrer une liste de widgets enfants (`children`). Il n'a pas vocation à créer un défilement horizontal mais juste à répartir ces enfants dans l'espace qui lui est donné. Il possède une propriété `mainAxisAlignment`, comme `Column`, dont les valeurs possibles sont identiques.

© Editions ENI - All rights reserved

Pour comprendre la mise en place, voici un autre extrait du code de la figure 37. Celui-ci fait référence aux colonnes d'icônes et montre leur intégration au sein d'une même ligne (Row) :

```
Row(
 mainAxisAlignment: MainAxisAlignment.spaceAround,
 children: <Widget>[
   Column(
     ...
   ),
   Column(
     ...
   ),
   Column(
     ...
   ),
   Column(
     ...
   ),
   Column(
     ...
   ),
   Column(
     ...
   ),
 ],
),
```

6.4.4 Expanded

Le widget Expanded est très intéressant notamment quand il est utilisé conjointement avec les lignes (Row) et les colonnes (Column). Son rôle est d'étendre, de modifier la proportion que prend un enfant dans une Column ou une Row. Pour cela, il possède une propriété nommée flex qui va servir de coefficient.

Dans une colonne possédant deux enfants, si les deux sont entourés d'un Expanded avec une valeur de flex respectivement à 1 pour le premier et à 2 pour le second, alors le deuxième enfant prendra deux fois plus de place dans la colonne que son frère.

Dans l'exemple présenté dans la figure 37, de haut en bas, dans la colonne principale, on peut distinguer au moins trois éléments importants : le titre (`Alignements :`), le texte des différents alignements possibles (`start`, `end`, `center`, ...) et les colonnes d'icônes. Visuellement, il est flagrant de voir que la partie du bas où se situent les icônes est beaucoup plus grande que les deux autres. C'est ce que produit un `Expanded`. Voici le code correspondant :

```
Column(
 mainAxisAlignment: MainAxisAlignment.spaceAround,
 children: <Widget>[
   Padding(
     padding: EdgeInsets.only(top: 20),
   ),
   Text(
     'Alignements :',
     style: TextStyle(
       fontSize: 30,
       fontWeight: FontWeight.bold,
       color: Colors.white,
     ),
   ),
   Padding(
     padding: EdgeInsets.only(top: 20),
   ),
   Expanded(
       flex: 4,
       child: Row(
         ...
       )
),
   Divider(
     color: Colors.white,
   ),
   Expanded(
     flex: 20,
     child: Row(
       ...
     ),
   )
 ],
),
```

© Editions ENI - All rights reserved

La zone des titres prend une valeur `flex` de 4 tandis que celle de la ligne des icônes est à 20. Sans ces valeurs, les lignes auraient eu chacune la même proportion. `Expanded` est donc un outil très pratique pour gérer la disposition sur un écran.

6.4.5 Container

Le Container est simplement un conteneur à taille défini dans lequel d'autres widgets peuvent venir se placer. Par défaut, sa dimension est celle de son enfant et comme il ne possède pas de couleur, il est invisible.

En revanche, il est possible de lui attribuer une couleur grâce à la propriété `color`. Il devient alors visible. L'avantage du Container est qu'il est possible de lui donner un style bien précis grâce à une propriété : `decoration`. Elle peut accepter des widgets `BoxDecoration` ou `ShapeDecoration`. Grâce à ceux-ci, il est possible de définir des bordures par exemple.

La dimension du Container est modulable avec plusieurs propriétés. Il est possible de jouer sur le `padding` qui correspond à la marge interne. De la même façon, il faudra utiliser `margin` pour gérer l'écart du container vis-à-vis des widgets qui l'entourent.

Enfin, il dispose de deux propriétés essentielles : `width` et `height`. Elles permettent de définir une taille pour la largeur et la hauteur.

Voici un premier exemple de Container :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(title),
   ),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Container(
           child: Text('Container 1'),
           color: Colors.blue,
           margin: EdgeInsets.only(bottom: 30),
         ),
```

```
    ],
  ),
),
);
}
```

Dans cet exemple, un Container contenant un texte est coloré en bleu. Une marge basse est mise en place afin de pouvoir accueillir d'autres widgets sans qu'ils soient collés les uns aux autres.

Comme le rendu le montre, le Container prend bien pour dimension son contenu :

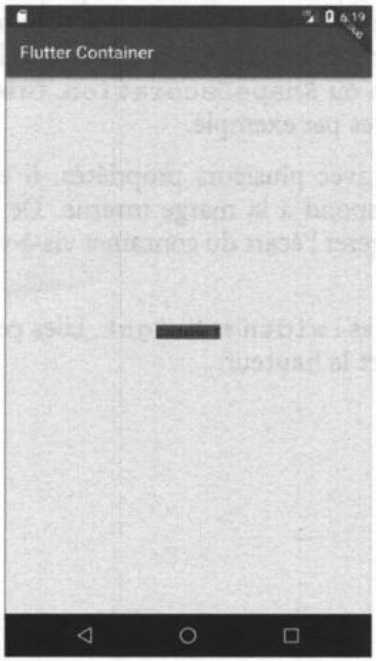


Figure 38 - Exemple de Container basique

© Editions ENI - All rights reserved

Un second exemple est ajouté. Cette fois-ci, ce conteneur est dimensionné précisément en largeur et en hauteur. Il est intéressant de constater que son contenu, en l'occurrence le texte, se retrouve obligé de s'adapter à cette nouvelle dimension. Avant de s'attarder sur le visuel, voici le code évoqué :

```
Container(
 child: Text('Container 2'),
 color: Colors.green,
 height: 40,
 width: 40,
 margin: EdgeInsets.only(bottom: 30),
),
```

Et le rendu :

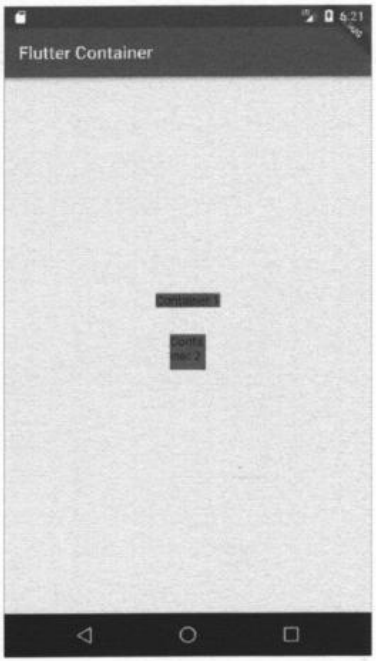


Figure 39 - Container dimensionné

Un troisième exemple explore les possibilités de la propriété `decoration` en utilisant le widget `BoxDecoration`. La forme donnée au Container n'est plus rectangulaire mais ronde. Par défaut et sans autre élément de contrainte, la dimension du conteneur se cale sur la plus petite valeur présente entre la hauteur et la largeur du widget enfant. Dans le cas du texte qui est contenu ici, la plus petite valeur est la hauteur évidemment.

```
Container(
 child: Text('Container 3'),
 decoration: BoxDecoration(
   shape: BoxShape.circle,
   color: Colors.red
 ),
 margin: EdgeInsets.only(bottom: 30),
),
```

Sur le rendu, le conteneur coloré en rouge est bien un cercle dont le diamètre correspond à la hauteur du texte :

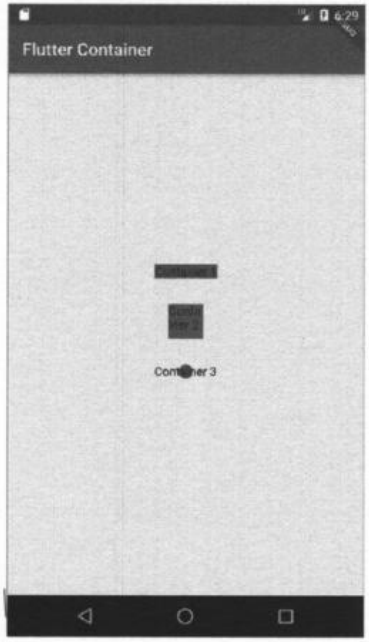


Figure 40 - Conteneur avec une decoration BoxDecoration

© Editions ENI - All rights reserved

L'exemple suivant repart du code précédent et le fait évoluer pour que le cercle rouge englobe la totalité du widget `Text`. Pour réaliser cette opération, il suffit de jouer sur la marge interne, le `padding` :

```
Container(
 child: Text('Container 4'),
 decoration: BoxDecoration(
     shape: BoxShape.circle,
     color: Colors.red
 ),
 padding: EdgeInsets.all(30),
 margin: EdgeInsets.only(bottom: 30),
),
```

Le résultat est un peu plus satisfaisant :

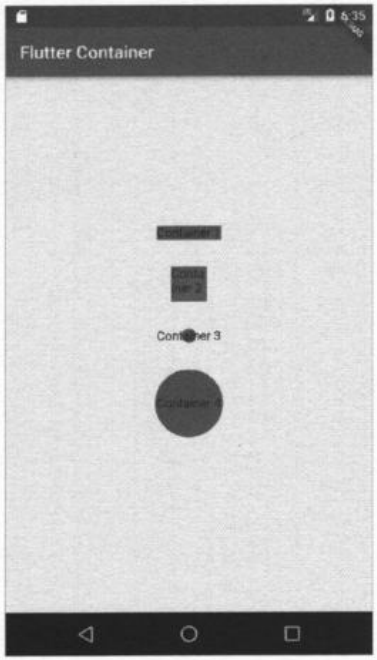


Figure 41 - Container avec une decoration BoxDecoration dimensionné

Enfin le dernier exemple va explorer le widget `ShapeDecoration` pour mettre en place une bordure à plusieurs couleurs.

```
Container(
 child: Text('Container 5'),
 padding: EdgeInsets.all(15),
 decoration: ShapeDecoration(
     color: Colors.white,
     shape: Border.all(
       color: Colors.red,
       width: 8,
     ) + Border.all(
       color: Colors.green,
       width: 8.0,
     )+ Border.all(
       color: Colors.blue,
       width: 8.0,
     ),
 ),
),
```

Le résultat est le suivant :

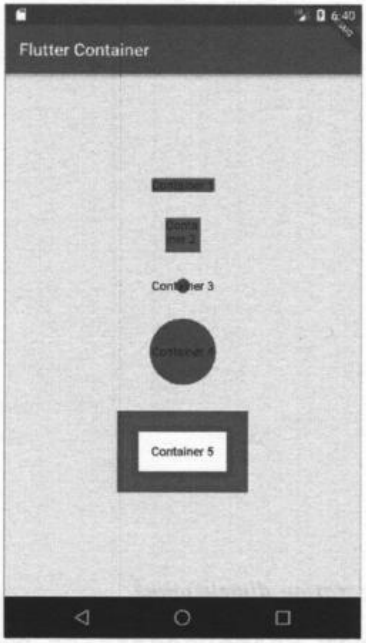


Figure 42 - Container avec des bordures de différentes couleurs

© Editions ENI - All rights reserved

Pour finir sur les conteneurs, il faut évoquer une dernière propriété intéressante : `alignment`. Elle permet de positionner avec précision l'enfant dans le `Container`. Il est possible de recourir à la classe `Alignment` qui propose plusieurs valeurs par défaut :

- `topLeft` (en haut à gauche)
- `topRight` (en haut à droite)
- `topCenter` (centré en haut)
- `bottomLeft` (en bas à gauche)
- `bottomCenter` (centré en bas)
- `bottomRight` (en bas à droite)
- `center` (centré)
- `centerLeft` (centré à gauche)
- `centerRight` (centré à droite)

Voici un exemple de son implémentation :

```
Container(
 height: 150,
 width: 150,
 color: Colors.blueGrey,
 child: Icon(
   Icons.directions_car,
   color: Colors.white,
   size: 40,
 ),
 alignment: Alignment.bottomRight,
),
```

Dans ce code, un `Container` formant un carré gris de 150 pixels de côté intègre une icône de voiture. Par défaut, elle se situe au centre de son widget parent.

En valorisant le paramètre `alignment` avec `Alignment.bottomRight`, elle vient se placer désormais en bas à droite :

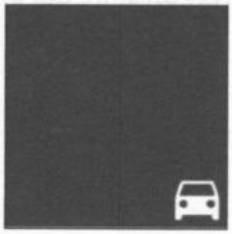


Figure 43 - Alignement d'un enfant dans un Container

Si les choix proposés par défaut ne conviennent pas, `Alignment`, via son constructeur, accepte aussi des valeurs précises pour les abscisses (x) et les ordonnées (y). Elles vont de -1 à 1 selon ce graphique

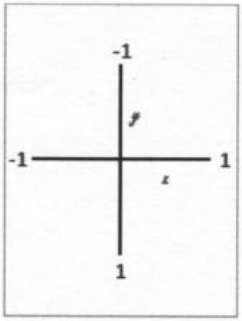


Figure 44 - Valeurs d'alignement

En reprenant le code précédent, il est possible de faire évoluer l'alignement de cette façon :

```
Container(
 height: 150,
 width: 150,
 color: Colors.blueGrey,
 child: Icon(
   Icons.directions_car,
   color: Colors.white,
   size: 40,
```

© Editions ENI - All rights reserved

```
  ),
  alignment: Alignment(-0.8,0.7),
),
```

Cette fois-ci, la position de la voiture est de -0.8 sur les abscisses donc près du bord gauche et de 0.7 pour les ordonnées, c'est-à-dire située assez bas.

Figure 45 - Alignement précis d'un enfant dans un Container

6.4.6 Padding

Le `Padding` est un widget qui a déjà été rencontré dans quelques exemples. Il sert à gérer un espace. Potentiellement, il peut aussi accueillir un enfant.

Il n'y a pas forcément beaucoup de différence entre ce widget et un `Container`. Toutefois, il est assez aisé de le mettre en place.

Dans le constructeur, il n'y a que deux paramètres attendus. Le premier, obligatoire, est le `padding` qui correspond à un objet héritant de la classe `EdgeInsetsGeometry`. Le second, facultatif, est le `child`, c'est-à-dire un widget enfant.

`EgdeInsetsGeometry` est une classe abstraite et ne peut donc être instanciée. En revanche, elle possède deux classes filles qui, elles, peuvent l'être : `EdgeInsets` et `EdgeInsetsDirectionnal`.

La classe `EdgeInsets` possède plusieurs constructeurs nommés qui viennent répondre à différents besoins :

- `EdgeInsets.all(double valeur)` : permet, avec une seule valeur, d'appliquer un écart à l'ensemble des côtés (haut, bas, gauche et droite).

Padding de 20 pixels pour chaque côté :

```
Padding(padding: EdgeInsets.all(20)),
```

- `EdgeInsets.fromLTRB(double gauche, double haut, double droite, double bas)` : permet de préciser un écart pour chaque côté.

Padding de 20 pixels à gauche, 10 en haut, 5 à droite et 30 en bas :

```
Padding(padding: EdgeInsets.fromLTRB(20, 10, 5, 30)),
```

- `EdgeInsets.only({double gauche: 0.0, double haut: 0.0, double droite: 0.0, double bas: 0.0})` : permet d'appliquer un écart à un ou plusieurs côtés. Les côtés non valorisés prendront la valeur 0.0 par défaut.

Padding de 10 pixels à gauche et 20 à droite (et donc 0 en haut et en bas) :

```
Padding(padding: EdgeInsets.only(left: 10, right: 20)),
```

- `EdgeInsets.symmetric({double vertical: 0.0, double horizontal: 0.0})` : Permet de préciser un écart vertical (haut et bas) et/ou horizontal (gauche et droite). Le paramètre non valorisé prendra la valeur 0.0 par défaut.

Padding vertical de 10 pixels et 30 en horizontal :

```
Padding(padding: EdgeInsets.symmetric(vertical: 10, horizontal: 30)),
```

La classe `EdgeInsetsDirectionnal` est basée sur les mêmes principes. La différence notable réside dans le fait qu'il n'y a plus ni gauche ni droite mais un début et une fin.

© Editions ENI - All rights reserved

6.4.7 Divider

Le `Divider` est un widget assez simple. Son rôle est de créer une ligne séparatrice horizontale. Il est couramment utilisé dans les `Drawer` ou les listes (voir chapitre Listes et grilles).

Son constructeur permet de travailler sur différentes propriétés comme la couleur du trait, son épaisseur, sa marge verticale ou encore son indentation.

Voici un exemple :

```
Divider(
 color: Colors.blue,
 indent: 40,
 endIndent: 40,
 thickness: 2,
 height: 30,
),
```

Dans ce code, on ajoute un `Divider` bleu (`color`). Son indentation par rapport à la marge gauche (`indent`) est de 40 pixels, tout comme l'indentation à droite (`endIntent`). L'épaisseur du trait (`thickness`) est accentuée un peu (par défaut sa valeur est 1). Enfin, l'espace global du `Divider` est augmenté.

Ce `Divider` a été ajouté à l'écran des `Container` afin de créer une séparation entre les deux derniers. À titre de comparaison, un second `Divider` configuré par défaut (aucun paramètre du constructeur n'est appelé) est inséré au-dessus.

Voici, le rendu :

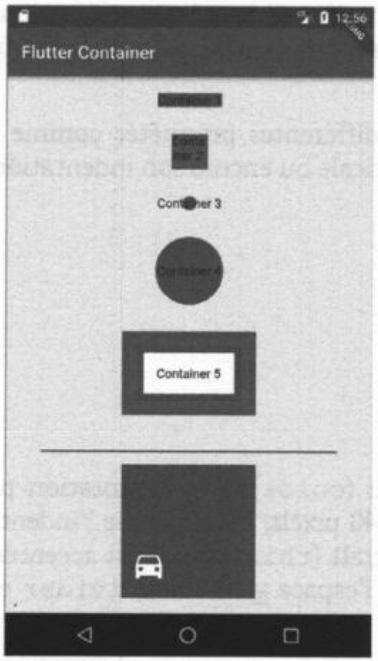


Figure 46 - Divider

6.4.8 Stepper

Le widget `Stepper` permet de gérer une séquence pas à pas. Le `Stepper` a vocation à évoluer au fil du temps.

Pour découvrir ce formidable outil, un nouveau projet est commencé. La classe principale étend `StatefulWidget` (sujet évoqué au chapitre Les états) afin d'être dynamique :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
```

© Editions ENI - All rights reserved

```
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Stepper'),
    );
  }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             'Stepper',
             style: Theme.of(context).textTheme.display1,
           ),
         ],
       ),
     ),
   );
 }
}
```

Le widget `Stepper` doit posséder a minima une liste d'étapes, d'objets `Step`. Dans la classe `_MyHomePageState`, cette liste est mise en place. La classe `Step` réclame deux éléments obligatoires : un titre (`title`) et un contenu (`content`) :

```
List<Step> steps = [
 Step(title: Text('Palier 1'), content: Text('Ceci est le contenu du Stepper 1')),
 Step(title: Text('Palier 2'), content: Text('Ceci est le contenu du Stepper 2')),
 Step(title: Text('Palier 3'), content: Text('Ceci est le contenu du Stepper 3')),
 Step(title: Text('Palier 4'), content: Text('Ceci est le contenu du Stepper 4')),
];
```

Le `Stepper` peut aussi avoir besoin qu'on lui indique l'index courant pour savoir sur quel item démarrer. Par défaut celui-ci est 0. Dans l'exemple, un entier va être ajouté, il va jouer ce rôle. Plus tard, il va servir à faire évoluer la progression du `Stepper`.

```
int _index = 0;
```

À partir de maintenant, les éléments de base sont présents pour incorporer le `Stepper`. Il est inséré juste en-dessous de `Text` :

```
Stepper(
 steps: steps,
 onStepContinue: _incrementStepper,
 onStepCancel: _decrementStepper,
 currentStep: _index,
),
```

Le premier paramètre du constructeur est `steps`. Il attend une liste de `Step`. Il suffit donc de lui donner la liste préalablement créée. Le second paramètre donné est l'indice courant. L'entier `_index` lui est passé.

Comme il s'agit d'un widget qui va évoluer au fil du temps, les étapes (`Step`) vont se succéder. Il possède pour chaque étape, sur laquelle il s'arrête, deux boutons par défaut : `continue` et `cancel`. Il convient donc de les configurer. Pour cela, il existe deux paramètres dans le constructeur, il s'agit respectivement de `onStepContinue` et `onStepCancel`. Il suffit donc de passer des méthodes qui vont indiquer le comportement qui va s'opérer lors de l'appui sur ces boutons. Pour continuer la progression, le clic sur continue appellera une méthode nommée `_incrementStepper`. Pour revenir en arrière, l'appui sur `cancel` appellera `_decrementStepper`.

© Editions ENI - All rights reserved

Voici le code de ces deux méthodes :

```
void _incrementStepper() {
 setState(() {
   _index++;
 });
}

void _decrementStepper() {
 setState(() {
   _index--;
 });
}
```

Si la physionomie de ces boutons ne convient pas, il est bien sûr possible de la changer. Pour cela, le constructeur de `Stepper` possède un paramètre intitulé `controlsBuilder`.

Il est temps d'avoir un aperçu du `Stepper` :

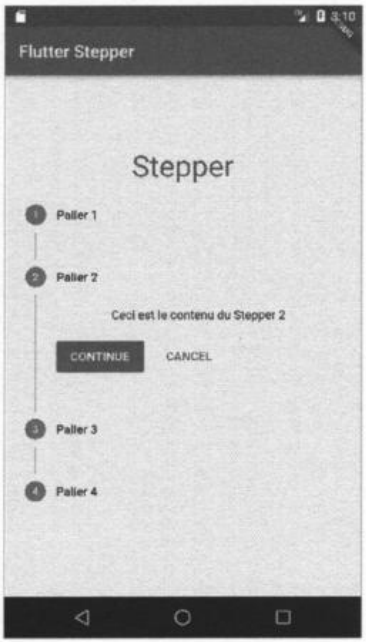
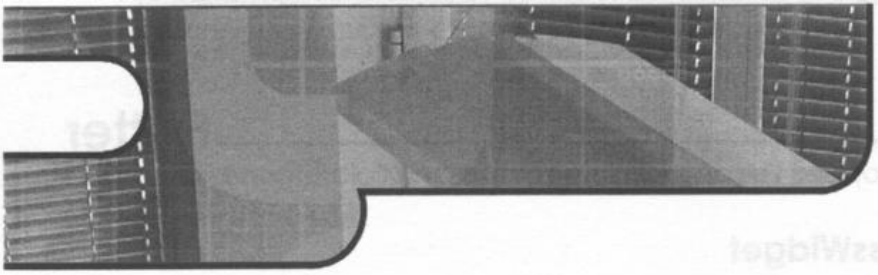


Figure 47 - Stepper

Durant toutes les démonstrations, deux éléments récurrents sont apparus, les `StatelessWidget` et `StatefulWidget`. Il est l'heure de découvrir en détail le système d'état que les widgets possèdent.

© Editions ENI - All rights reserved



Chapitre 7
Les états

1. Introduction

La gestion des états est un point fondamental. Dans les divers exemples parcourus jusqu'à présent et à venir, deux héritages sont récurrents, `StatelessWidget` et `StatefulWidget`. Ils permettent de définir le comportement des classes ou des widgets. La question est de savoir s'ils auront besoin de voir leur état évoluer avec le temps ou non.

Cette gestion des états qui se retrouve dans les `StatefulWidget` est vraiment pratique, mais présente un point faible indéniable. La partie graphique de l'application est intimement liée à la partie logique. Ce qui rend difficilement maintenable une application. La bonne façon de procéder est de réussir à réduire ce couplage au maximum. Sur ce sujet, qui reste très vaste et sur lequel la position des développeurs reste mitigée, le choix, est ici fait de s'arrêter sur deux méthodes très répandues et reconnues, bien qu'il en existe d'autres. Dans un premier temps, les *providers* seront abordés. Ils sont aujourd'hui, selon les équipes de Flutter, la voie à privilégier. Toutefois, la communauté reste très attirée par le *pattern* BLoC. C'est donc un thème qu'il faut aborder aussi.

Bien sûr, ces sujets sont très complexes. Ce chapitre ne pourra être considéré que comme une première approche. En effet, s'il fallait rentrer dans le détail et découvrir l'ensemble des pratiques, il faudrait y consacrer un ouvrage entier.

2. StatelessWidget

Le `StatelessWidget` est une classe (un widget) qui sert à créer une interface utilisateur. Comme son nom l'indique, elle n'a pas vocation à voir son état évoluer dans le temps.

Une classe qui implémente `StatelessWidget` doit impérativement intégrer un widget `build` :

```
class MyHomePage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   // TODO: implement build
   return null;
 }
}
```

Ce `build` est appelé à plusieurs moments. D'abord, quand le widget est inséré dans l'arbre des widgets, quand son parent subit une modification ou bien quand il est associé à un autre widget `InheritedWidget`.

Les deux derniers appels, qui peuvent arriver régulièrement, vont forcer le widget à se reconstruire, diminuant ainsi la performance de l'application. Il existe cependant plusieurs points à respecter pour minimiser ce phénomène.

La première méthode consiste, si le contexte le permet, à minimiser le nombre de widgets en son sein. Notamment, si un seul élément doit être placé, il convient d'éviter d'utiliser les `Row`, `Column`, `Padding` et `SizedBox`. Dans ce cas, il sera préférable de recourir à `Align` ou à `CustomChildLayout`. De la même façon, plutôt que d'imbriquer plusieurs couches de `Container` (avec leur `Decoration`) pour obtenir un effet graphique particulier, il vaudra mieux privilégier un simple `CustomPaintWidget`.

Dans la mesure du possible, il est très recommandé d'utiliser les constantes (`const`) pour les widgets ainsi que pour les constructeurs :

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
```

© Editions ENI - All rights reserved

```
      home: MyHomePage(title: 'Stateless', color: Colors.orange,),
    );
  }
}

class MyHomePage extends StatelessWidget {
 const MyHomePage({Key key, this.title, this.color}) : super(key: key);

 final Color color;
 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       backgroundColor: color,
       title: Text(title),
     ),
   );
 }
}
```

Il est aussi possible de basculer le widget en `StatefulWidget` pour bénéficier de certains aspects tels que la mise en cache des parties communes des sous-arbres ou l'utilisation des `GlobalKeys` lors des changements de structure de l'arbre.

Enfin, pour limiter les reconstructions d'un widget qui hérite de `StatelessWidget` en raison de l'utilisation de `InheritedWidget`, il convient de le refactoriser tant que possible en plusieurs widgets. Cela permet ainsi d'individualiser les reconstructions, ce qui est moins coûteux en performance.

3. StatefulWidget

Le `StatefulWidget` est, a contrario du `StatelessWidget`, un widget qui a vocation à voir son état évoluer avec le temps.

Dès lors, sa construction est un peu différente et se réalise par le biais de deux classes. La première, classique, étend `StatefulWidget`. Elle a un rôle plutôt déclaratif. C'est donc une classe qui va contenir un constructeur et des attributs par exemple. Elle contient aussi, et c'est obligatoire, l'appel à la méthode `createState`. Cette dernière va retourner une instance d'une classe gérant le `State` par l'appel à son constructeur. Elle prendra en charge deux choses. Premièrement, elle décrira des widgets, principalement des éléments graphiques. Deuxièmement, elle sera capable de les faire changer et évoluer par le recours à une méthode `setState`.

Voici un exemple de classe étendant `StatefulWidget` :

```
class Accueil extends StatefulWidget {
 Accueil({Key key, this.title, this.color}) : super(key: key);

 final String title;
 Color color;

 @override
 State<StatefulWidget> createState() {
   return _AccueilState();
 }
}
```

Il s'agit d'une construction assez classique. Toutefois, la méthode `createState` peut être optimisée et il est préférable de la retrouver sous cette syntaxe :

```
@override
_AccueilState createState() => _AccueilState();
```

Concernant la classe `_AccueilState`, elle va devoir prendre en charge la partie graphique ainsi que son évolution potentielle au travers de `setState`. Pour offrir un autre exemple que celui habituellement donné avec l'application par défaut, une variante est proposée. Cette fois, l'appui sur le `FloatingActionButton` va modifier aléatoirement la couleur de divers éléments de l'écran.

© Editions ENI - All rights reserved

Cette classe doit étendre `State<T>`. T correspondant à la classe associée étendant le `StatefulWidget`. Dans l'exemple, T sera remplacé par `Accueil`.

```
class _AccueilState extends State<Accueil> {
```

Dans l'écran principal, il y aura une `AppBar`, un bouton flottant et un `Container` affichant les valeurs RGB de la couleur.

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: widget.color,
   ),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'La couleur affichée est :',
         ),
         Container(
           width: MediaQuery.of(context).size.width * 0.8,
           height: MediaQuery.of(context).size.height * 0.13,
           color: widget.color,
           child: Center(
             child: Text('R(${widget.color.red}),
G(${widget.color.green}), B(${widget.color.blue})',
               textAlign: TextAlign.center,
               style: TextStyle(
                 color: Colors.white,
                 fontSize: 30,
                 fontWeight: FontWeight.bold
               ),
             ),
           ),
         ),
       ],
     ),
   ),
   floatingActionButton: FloatingActionButton(
     onPressed: _changementCouleur,
     tooltip: 'Increment',
```

```
      child: Icon(Icons.color_lens),
      backgroundColor: widget.color,
    ),
  );
}
```

Dans le code qui précède, il faut noter que l'attribut `color` de la classe `Accueil` va être utilisé. Pour l'appeler ici, il suffit de recourir à `widget` avec la syntaxe `widget.color`. Il en est de même pour le titre dans l'`AppBar`, `widget.title`.

Pour afficher les valeurs RGB de `color`, il existe trois fonctions exploitables sur les instances de `Color` : `red`, `green` et `blue`.

Le bouton d'action flottant appelle une méthode nommée `_changementCouleur`. L'objectif de cette dernière est de réaliser le tirage au sort d'une couleur. Pour cela, trois tirages sont réalisés, un pour chaque valeur du RGB, soit un tirage pour le rouge, un pour le vert et un pour le bleu. Pour rappel, ces valeurs doivent être comprises entre 0 et 255. Afin de faciliter cette tâche, trois variables sont créées au début de la classe `State` :

```
int _red = null;
int _green = null;
int _blue = null;
```

L'obtention d'une valeur aléatoire passe par la classe `Random`. Une fois les valeurs tirées au sort, un appel à `setState` est exécuté. C'est l'élément fondamental ici. C'est d'ailleurs pour cela que la classe est un `StatefulWidget`.

```
void _changementCouleur() {
  _red = Random.secure().nextInt(256);
  _green = Random.secure().nextInt(256);
  _blue = Random.secure().nextInt(256);
  setState(() {
    widget.color = Color.fromRGBO(_red, _green, _blue, 1);
  });
}
```

© Editions ENI - All rights reserved

Dans le cas présent, `setState` va mettre à jour la variable `color`. De facto, tous les éléments du `build` qui contiennent cette variable vont devoir se réactualiser également. L'affichage va donc être réactif. Pour être performant, il convient d'inclure dans `setState` uniquement l'évolution des données et d'éviter d'y incorporer les traitements logiques. C'est pour cela que les tirages au sort se déroulent avant et que le `setState` ne gère que la mise à jour de `widget.color`.

Voici à quoi ressemble l'application terminée :

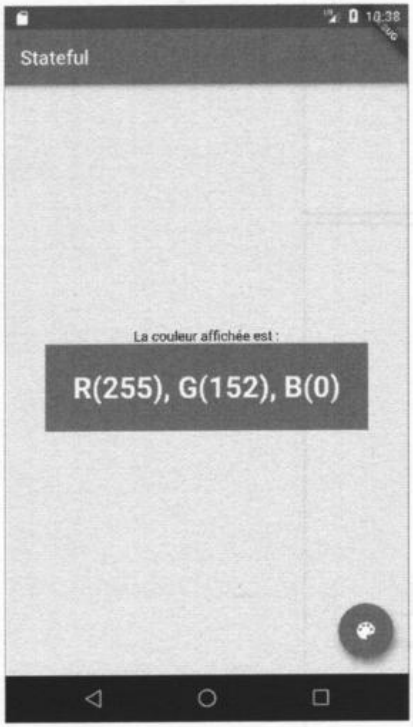


Figure 1 - StatefulWidget

3.1 Cycle de vie

L'application développée ci-dessus, avec l'utilisation de `setState`, est une bonne entrée en matière sur les `StatefulWidget`. En effet, c'est l'occasion de comprendre le cycle de vie de ces objets.

Voici un schéma détaillé du cycle de vie d'un `StatefulWidget` (les différentes méthodes seront expliquées un peu après) :

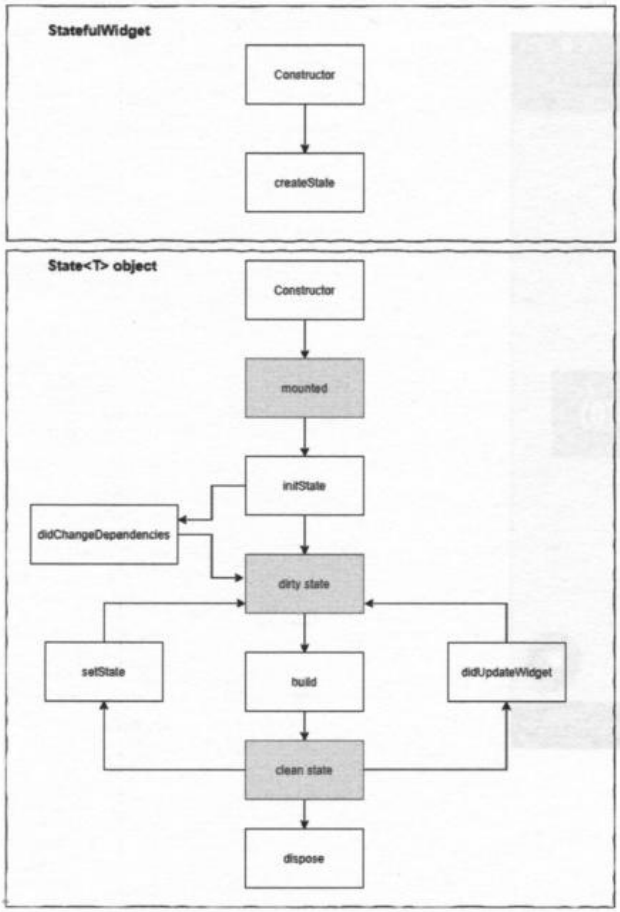


Figure 2 - Cycle de vie d'un StatefulWidget et de son objet State

© Editions ENI - All rights reserved

- `createState` : cette méthode permet, comme son nom l'indique, de créer l'état associé au `StatefulWidget`.
- `mounted` : il s'agit d'un booléen qui correspond au moment où le `State` est monté. C'est-à-dire qu'un `BuildContext` lui est associé, qu'il prend place dans l'arbre de widgets.
- `initState` : dans l'objet `State`, c'est la première méthode qui sera appelée. Elle permet d'initialiser des données par exemple. Dans tous les cas, elle doit faire référence au constructeur de la classe mère par `super.initState()`. Cette méthode ne s'exécute qu'une seule fois.
- `didChangeDependencies` : cette méthode est appelée quand une dépendance de la classe de l'objet `State` est modifiée. Elle s'exécute une fois, directement après `initState`. Cependant, elle pourra être appelée plus tard au besoin. Ce sera notamment le cas, si ce widget est associé à un `InheritedWidget`.
- `dirty` : en français, dirty signifie « sale ». Avec Flutter, un objet `State` considéré comme sale a besoin d'être reconstruit.
- `build` : l'objet `State` va être construit ou reconstruit lors de cette étape.
- `clean` : l'objet `State` est construit et « propre ». Deux actions peuvent pousser ce widget à revenir à un état sale (`dirty`) forçant sa reconstruction : `setState` et `didUpdateWidget`. Il convient aussi de préciser que le widget pourra être reconstruit si, étant associé à un `InheritedWidget`, la méthode `didChangeDependencies` est appelée.
- `setState` : cette méthode a déjà été abordée. Elle permet de remettre le widget dans un état sale (`dirty`) et provoque donc de nouveau l'exécution de `build`. Cela concerne les changements internes de l'objet.
- `didUpdateWidget` : si un widget ancêtre dans l'arbre des widgets prévoit que ce widget soit reconstruit, alors il exécutera cette méthode. Mais il faut que cette reconstruction se fasse avec le même type de widget et que la clé (`key`) soit identique.
- `dispose` : cette méthode permet de supprimer de manière permanente le widget de l'arbre des widgets.

4. Couplage interface et logique

Le cycle de vie des `StatefulWidget` est complexe mais offre beaucoup de possibilités. En particulier les méthodes `setState` qui sont extrêmement pratiques comme les diverses démonstrations l'ont prouvé.

En revanche, il reste un problème d'envergure. L'inclusion d'un `setState` comprenant de la logique applicative au sein même de l'interface graphique rend l'application difficilement maintenable. De plus, un `setState` étant lié à un contexte (`context`), il n'est pas possible de faire évoluer plusieurs interfaces en même temps.

De manière générale, en programmation, la bonne pratique consiste à séparer la partie logique de la gestion graphique. Concernant Flutter, il existe de nombreuses méthodes permettant de mettre en place cette séparation. Sans être exhaustif, en voici quelques-unes :

- Provider
- BLoC
- InheritedWidget
- ScopedModel
- Redux
- MobX
- ...

Le choix est donc difficile et le débat n'est pas clos en la matière. Il faut bien être conscient qu'il s'agit là de notions avancées qui mériteraient un ouvrage à elles seules si elles devaient être toutes détaillées.

Toutefois, dans la liste ci-dessus, il y a tout de même deux pratiques qui se détachent : les *providers* et les BLoC. Les sections suivantes n'entreront pas dans le détail, mais elles permettront d'en comprendre les principes de base.

À cet effet, la démonstration de lancement (le compteur) sera retravaillée avec les deux méthodes. Elle a l'avantage d'être simple et surtout, elle a déjà été manipulée à plusieurs reprises.

© Éditions ENI - All rights reserved

4.1 Provider

Les *providers* peuvent être considérés comme la pratique de référence. En effet, lors de la conférence Google IO de 2019, c'est cette voie qui a été considérée comme étant celle à privilégier.

Pourquoi ?

Les raisons sont simples : cela fonctionne très bien et l'approche est plus aisée qu'avec d'autres méthodes.

L'avantage va donc être de pouvoir travailler en utilisant `StatelessWidget`. Il n'y aura pas de gestion d'état dans la partie graphique de l'application.

La partie logique est donc externalisée et peut resservir à un autre endroit du programme au besoin.

Le couplage entre l'interface et la logique de l'application est donc faible.

Pour faciliter son implémentation, Google préconise l'utilisation d'une dépendance. Il s'agit de **provider** en version 4.0.4 au moment où ces lignes sont écrites. Voici le lien qui permet d'y accéder :
https://pub.dev/packages/provider#-readme-tab-

Le package est estampillé par le badge *Flutter Favorite*. Il s'agit d'un programme visant à identifier les packages ou les plugins qui devraient être favorisés lors de la création d'applications. Ces packages doivent répondre à un certain nombre de critères et exigences. Ce titre est attribué par un comité composé de membres de l'équipe Flutter et de la communauté. À ce jour, 24 packages ont reçu cette récompense.

Figure 3 - Badger Flutter Favorite

Pour utiliser ce package, un nouveau projet est mis en place. Comme cela a été indiqué, l'application de base du compteur est réutilisée. Seulement, le fichier *main.dart* est un peu modifié pour que la classe `MyHomePage` étende `StatelessWidget` et non plus `StatefulWidget`.

Mais avant de s'attaquer à la partie graphique, il va falloir importer le package **provider**. Dans le fichier *pubspec.yaml*, il faut ajouter sous `dependencies` :

```
provider: ^4.0.4
```

Attention, il ne faut pas oublier de cliquer sur le lien **Packages get** pour récupérer ces ressources.

La suite va intervenir dans un nouveau fichier *.dart* qui va être créé dans le dossier *lib*. Dans cet exemple, ce fichier est nommé *compteur_provider.dart*.

La première chose à y faire est d'importer le package **material** puisqu'il va évidemment servir ici.

```
import 'package:flutter/material.dart';
```

Puis, une classe nommée `CompteurProvider` est mise en place. Elle va utiliser un *mixin* (cf. le chapitre Dart, notions avancées) de la classe `ChangeNotifier`. Cette classe provient du package **material**. Afin de réaliser cela, il faut utiliser le mot-clé `with`.

```
class CompteurProvider with ChangeNotifier {

}
```

Cette classe va donc contenir la logique du programme. Dans l'exemple du compteur, la logique se résume à un entier qui va s'incrémenter. C'est tout !

Dans `CompteurProvider`, il va falloir retrouver ces éléments. D'abord, un entier :

```
int _compteur;
```

© Editions ENI - All rights reserved

Cette variable est privée, il est important de mettre en place aussi un constructeur et les *getters* et *setters* qui y sont associés.

```
CompteurProvider(this._compteur);

int get counter => _compteur;

set counter(int value) {
 _compteur = value;
}
```

Le constructeur va permettre d'indiquer une valeur de départ à l'entier. Le *getter* va permettre de lire la valeur qui s'y trouve.

Il ne reste plus que l'incrémentation à gérer. Il faut créer une méthode qui va gérer cela :

```
void increment() {
 _compteur++;
}
```

Il ne manque plus qu'une dernière chose. Pour être efficace, ce système va avoir besoin de prévenir l'interface que l'incrémentation a eu lieu. Il faut prévoir cette fonctionnalité. Le package **material** a justement tout prévu. Il existe une méthode `notifyListeners` qui va se charger de prévenir qu'un changement s'est produit.

La méthode `increment` évolue et devient :

```
void increment() {
 _compteur++;
 notifyListeners();
}
```

Afin d'être le plus clair possible et pour ceux qui découvrent notamment, voici le code entier de la classe `CompteurProvider` :

```
import 'package:flutter/material.dart';

class CompteurProvider with ChangeNotifier {
 int _compteur;

 CompteurProvider(this._compteur);
```

```
  int get counter => _compteur;

  set counter(int value) {
    _compteur = value;
  }

  void increment() {
    _compteur++;
    notifyListeners();
  }

}
```

Bien, maintenant que la logique est en place, il faut s'attaquer à la partie graphique de l'application. Visuellement, elle aura la même apparence que d'habitude.

L'important va être de signaler à cette interface qu'elle doit être à l'écoute de ce qui se passe dans la classe `CompteurProvider`. L'appel à `runApp` va lancer `MyApp`, comme d'habitude. Mais cette classe habituellement assez sommaire va ici servir à l'abonnement au provider.

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: ChangeNotifierProvider(
       create: (_) => CompteurProvider(0),
       child: MyHomePage(title: 'Flutter Provider'),
     )
   );
 }
}
```

© Editions ENI - All rights reserved

Dans la propriété home qui appelle généralement MyHomePage, ici, c'est une classe du package **provider** qui est utilisée, ChangeNotifierProvider. Ce constructeur va permettre de renseigner deux propriétés create et child.

La première va permettre d'appeler le constructeur du provider CompteurProvider. En paramètre la valeur 0 est indiquée pour le départ du compteur.

La syntaxe est un peu étrange puisque le paramètre de la fonction est un *underscore* (tiret bas).

```
(_) => ...
```

Le tiret bas correspond juste à une variable. Il pourrait être remplacé par n'importe quoi (x, maVariable...). Cette pratique est courante ainsi que cette nomenclature quand on prévoit de ne pas utiliser la variable en question. Il faut noter qu'il n'est pas possible de laisser les parenthèses vides, ce qui correspondrait à l'absence de paramètre.

Le ChangeNotifierProvider est maintenant rattaché au CompteurProvider. Dès lors, il sera averti dès que la méthode notifyListeners sera exécutée. Pour être précis, il imposera à tous ses descendants de se reconstruire à ce moment-là.

Le second paramètre, child, permet de reprendre le cours normal des choses et va appeler le constructeur de la classe MyHomePage.

MyHomePage va donc étendre, comme cela a été dit, StatelessWidget.

```
class MyHomePage extends StatelessWidget {
 MyHomePage({Key key, this.title}) : super(key: key);
 final String title;

 @override
 Widget build(BuildContext context) {
...
```

Dans le `build`, afin de pouvoir récupérer la valeur du compteur qui se trouve dans la partie logique, il faut mettre en place une variable :

```
final _compteurProvider = Provider.of<CompteurProvider>(context);
```

`Provider.of<T>(context)` permet d'accéder à l'objet `CompteurProvider`, et donc à tout ce qu'il contient, y compris la valeur de compteur.

Dans l'instruction `return`, il suffit maintenant de construire l'application comme à l'habitude. Partout où c'est nécessaire, il sera possible de récupérer la valeur du compteur grâce à :

```
_compteurProvider.counter
```

Le compteur va être placé dans l'`AppBar` ainsi que dans du texte au sein du `body` :

```
return Scaffold(
  appBar: AppBar(
    title: Text('$title - ${_compteurProvider.counter}'),
  ),
  body: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          'Valeur du compteur :',
        ),
        Text(
          '${_compteurProvider.counter}',
          style: Theme.of(context).textTheme.display1,
        ),
      ],
    ),
  ),
```

© Editions ENI - All rights reserved

Il reste enfin à intégrer le `FloatingActionButton`. À l'appel de `onPressed`, il pourra s'appuyer sur la méthode `increment` de `CompteurProvider` :

```
floatingActionButton: FloatingActionButton(
 onPressed: _compteurProvider.increment,
 tooltip: 'Increment',
 child: Icon(Icons.add),
),
```

Le rendu final est exactement identique à celui de l'application originale :

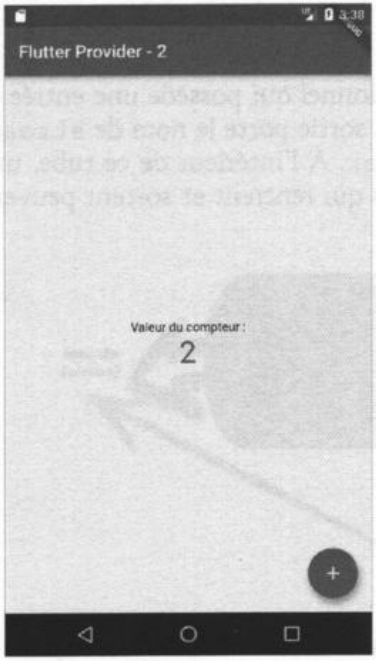


Figure 4 - Mise en place de Provider

Concernant les *providers*, il y aurait encore beaucoup à dire, notamment sur les `MultiProviders`. Le sujet est assez vaste. Mais ce qu'il faut retenir c'est la facilité de mise en place et l'efficacité. À l'heure actuelle, c'est certainement la manière à privilégier.

4.2 BloC

Une autre façon de procéder est de travailler avec des BLoC.

BLoC signifie ***B**usiness **L**ogic **C**omponent*. Il s'agit d'un *design pattern* ou patron de conception, c'est-à-dire une façon réfléchie et éprouvée de répondre à une problématique. Dans le cas qui nous concerne, c'est bien la séparation des couches logique et graphique dont il s'agit.

Dans l'exemple qui va suivre, l'implantation se fera à travers des objets `Stream`.

Qu'est-ce qu'un Stream ?

Un Stream est une sorte de tuyau unidirectionnel qui possède une entrée et une sortie. L'entrée est dénommée `sink`. La sortie porte le nom de `stream`. Le tout est géré par un `StreamController`. À l'intérieur de ce tube, une logique métier peut s'appliquer. Les données qui rentrent et sortent peuvent être de n'importe quel type sans restriction.

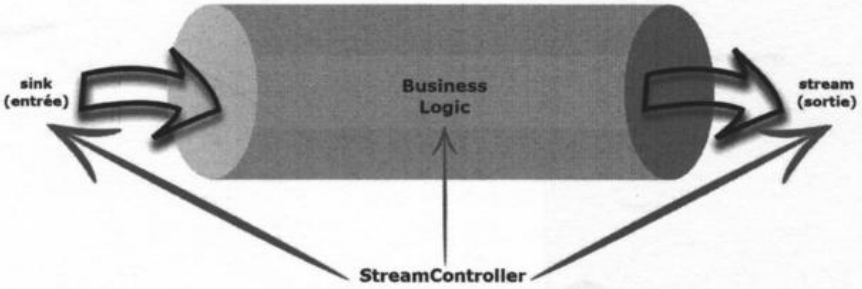


Figure 5 - Stream

En guise d'exemple, l'application du compteur va être de nouveau adaptée.

Pour utiliser les *streams*, il n'y a pas de package à charger. En effet, le `StreamController` sur lequel le programme s'appuie est déjà inclus dans le package `async` de Dart.

La première étape va donc consister à créer un nouveau fichier nommé *compteur_bloc.dart* dans le dossier *lib*. À l'intérieur, il faut créer une classe, nommée ici `CompteurBloc`.

© Editions ENI - All rights reserved

La première étape est d'obtenir une instance de `StreamController` :

```
import 'dart:async';

class CompteurBloc {

 final _streamCtrl = StreamController<int>();

}
```

L'appel au constructeur de `StreamController` permet de préciser le type de donnée que le *stream* va prendre en charge. Dans l'exemple du compteur, il s'agit d'entiers.

La seconde étape, fondamentale, est de penser à intégrer une méthode intitulée `dispose`. Le *stream* est un objet complexe et potentiellement lourd. Il convient dans le cadre de son cycle de vie de prévoir une méthode qui permettra de le fermer.

```
dispose() => _streamCtrl.close();
```

Ensuite, afin de contrôler les deux aspects du *stream*, il faut mettre en place les actions d'entrée et de sortie. Celles-ci sont réalisées à partir du `StreamController` bien entendu.

L'entrée est un objet de type `Sink`. Il est possible de restreindre le type des données entrantes comme dans l'exemple où seulement des entiers sont attendus.

```
Sink<int> get sink => _streamCtrl.sink;
```

Pour la sortie, le principe est le même. Cette fois-ci, le type est `Stream` :

```
Stream<int> get stream => _streamCtrl.stream;
```

Maintenant que l'ensemble des éléments permettant de faire fonctionner le *stream* sont en place. Il faut s'attaquer à la logique métier. L'objectif est de gérer le compteur à cet endroit. Il faut au préalable se munir d'une variable sur laquelle le code va s'appuyer :

```
int _compteur = 0;
```

Puis, il faut intégrer une méthode qui va être en charge de l'incrémentation. Cette dernière a un double rôle puisqu'elle doit non seulement incrémenter la variable `_compteur` mais aussi faire le lien avec le *stream*. Il faut que cette variable soit injectée via l'objet `sink` dans le *stream*.

```
IncrementCompteur() {
 sink.add(++_compteur);
}
```

Dans la mesure où l'incrémentation et l'ajout à l'objet `sink` ont lieu simultanément, il est nécessaire de pré-incrémenter la variable.

Voici la classe et son code au complet :

```
import 'dart:async';

class CompteurBloc {

 final _streamCtrl = StreamController<int>();
 int _compteur = 0;

 Sink<int> get sink => _streamCtrl.sink;
 Stream<int> get stream => _streamCtrl.stream;

 incrementCompteur() {
   sink.add(++_compteur);
 }

 dispose() => _streamCtrl.close();

}
```

Il convient maintenant de s'attaquer à la partie graphique. Contrairement à ce qui a été fait avec les *providers*, ici, la classe `MyHomePage` étendra `StatefulWidget`. Le but est de pouvoir profiter du cycle de vie et en particulier de `dispose`. Mais comme prévu, il n'y aura pas besoin de recourir au `setState`.

© Editions ENI - All rights reserved

La première partie du code est standard et ne contient aucun élément notable. Dans le State de MyHomePage, en revanche, il convient de récupérer une instance du BLoC CompteurBloc :

```
class _myHomePage extends State<MyHomePage> {

 final CompteurBloc _compteurBloc = CompteurBloc();

 @override
 Widget build(BuildContext context) {
     ...
```

Afin de pouvoir connaître la valeur du compteur dans le texte de l'écran, mais aussi dans l'AppBar, le retour du build va utiliser un StreamBuilder. Ce widget permet, comme son nom l'indique, de construire un *stream* assez facilement. Il est possible de lui indiquer également quel type sera pris en charge par ce *stream* :

```
return StreamBuilder<int>(
```

Ce constructeur permet de renseigner plusieurs paramètres.

D'abord l'objet stream afin de rester à l'écoute de notre BLoC.

```
stream: _compteurBloc.stream,
```

Il est possible d'initialiser la valeur de départ de la donnée via initialData. Dans le cas du compteur, l'initialisation se fait à 0 :

```
initialData: 0,
```

Si ce paramètre n'est pas renseigné, au démarrage, l'écran affiche null. Il est possible de contrer cela en créant un constructeur dans CompteurBloc où la valeur de compteur est transmise à l'objet sink :

```
CompteurBloc() {
 sink.add(_compteur);
}
```

Il est plus simple, dans notre cas, de laisser en place le paramètre `initialData` et de ne pas recourir au constructeur. Le code sera plus léger.

Enfin, le dernier paramètre qui va servir est `builder`. Voici le code qui va être détaillé juste après :

```
builder: (context, snapshot) {
 return ...
```

Deux paramètres vont être utiles, le contexte (`context`) et surtout `snapshot` qui correspond à une photographie à un instant T des données du *stream*.

L'instruction `return` permet enfin de placer le code standard de l'interface graphique, `Scaffold`, `Center`, `Column`...

Chaque fois où cela sera nécessaire, il sera possible de lire les données contenues dans `snapchot` avec sa propriété `data` :

```
snapchot.data
```

Voici le code complet du `State` avec les widgets graphiques :

```
class _myHomePage extends State<MyHomePage> {
 final CompteurBloc _compteurBloc = CompteurBloc();

 @override
 Widget build(BuildContext context) {
   return StreamBuilder<int>(
       stream: _compteurBloc.stream,
       initialData: 0,
       builder: (context, snapshot) {
         return Scaffold(
           appBar: AppBar(
             backgroundColor: Colors.red,
             title: Text('${widget.title} - ${snapshot.data}'),
           ),
           body: Center(
             child: Column(
               mainAxisAlignment: MainAxisAlignment.center,
               children: <Widget>[
                 Text('Valeur du compteur :'),
                 Text(
                   '${snapshot.data}',
```

© Editions ENI - All rights reserved

```
                    style: TextStyle(
                        color: Colors.red,
                        fontSize: 35,
                        fontWeight: FontWeight.bold),
                  )
                ],
              ),
            ),
            floatingActionButton: FloatingActionButton(
              child: Icon(Icons.add),
              onPressed: () => _compteurBloc.incrementCompteur(),
              backgroundColor: Colors.red,
            ),
          );
        });
  }

  @override
  void dispose() {
    _compteurBloc.dispose();
    super.dispose();
  }
}
```

Dernier point, la fin du `State` permet l'appel de `dispose` puisque cela fait partie de son cycle de vie. À l'intérieur, il est possible d'appeler le `dispose` du bloc. Cette opération n'aurait pas été envisageable avec un `StatelessWidget` qui ne possède pas ce cycle et de `dispose`.

Voici le rendu :

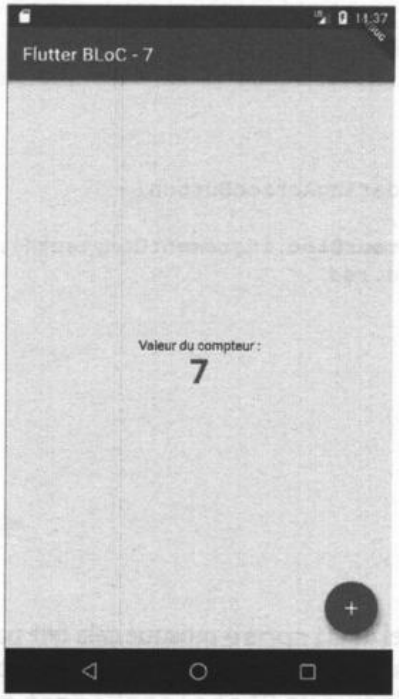


Figure 6 - Mise en place de Stream et Bloc

Pour terminer sur les BLoC, il faut noter l'existence de packages qui viendront faciliter la mise en place de ce *pattern*.

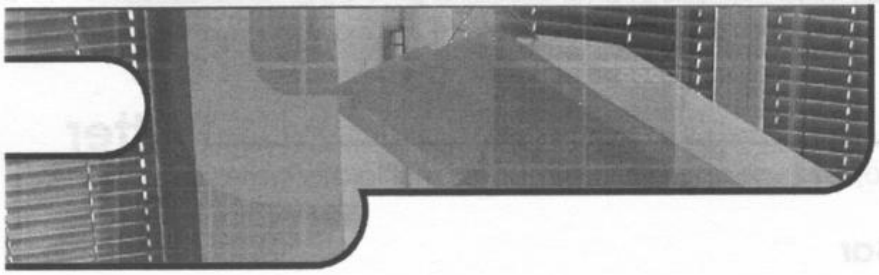
Dans les plus utilisés par la communauté, trois se distinguent :

flutter_bloc : https://pub.dev/packages/flutter_bloc avec une note globale de 100.

bloc : https://pub.dev/packages/bloc avec une note globale de 99.

bloc_pattern : https://pub.dev/packages/bloc_pattern avec une note globale de 98.

© Editions ENI - All rights reserved



Chapitre 8
Pop-up et navigation

1. Introduction

Ce chapitre est l'occasion de découvrir de nouveaux widgets. Dans cette partie, divers éléments de présentation seront découverts. Ils concernent des informations complémentaires qu'il peut être pratique d'afficher à l'écran.

Dans un premier temps, le widget `SnackBar` sera décrit. Il permet d'afficher un message passant à l'utilisateur.

Ensuite, d'autres affichages de type « modal » seront abordés. D'abord, pour les messages d'alerte à destination de l'utilisateur puis pour de simples informations.

Dans la continuité, les feuilles de choix qui offrent à l'utilisateur plusieurs solutions seront présentées.

Enfin, puisque les affichages sont évoqués tout au long de ce chapitre, il sera temps de s'attarder sur la navigation de page en page avec les routes.

2. SnackBar

`SnackBar` est un message passant qui pourrait s'apparenter à un « toast » bien connu sous Android. Le constructeur de `SnackBar` attend plusieurs paramètres. Un seul est obligatoire, `content`. Il s'agit bien sûr du contenu qui est, le plus souvent, un texte.

Il est possible de contrôler sa durée d'apparition grâce à la propriété `duration`. Comme pour de nombreux widgets, il est aussi possible de modifier sa couleur de fond avec `backgroundColor`.

Pour l'enrichir, un bouton interne peut être mis en place. En fonction des besoins, des actions peuvent y être associées. Ceci passe par le paramètre `action` alimenté par une instance de `SnackBarAction`. Son constructeur permet l'insertion d'un `label`, d'un `textColor` et évidemment d'un `onPressed`.

L'apparition du widget `snackBar` est déclenchée par l'appel à :

```
Scaffold.of(context).showSnakBar(nomDeLaSnackBarQuiDoitApparaitre)
```

Attention, la méthode `of` appelle l'objet `context` du `Scaffold`. Pour que cela soit possible, il convient de se trouver dans un nouveau contexte enfant. Sans nouveau contexte, une erreur se produira.

© Editions ENI - All rights reserved

Voici schématiquement le processus :

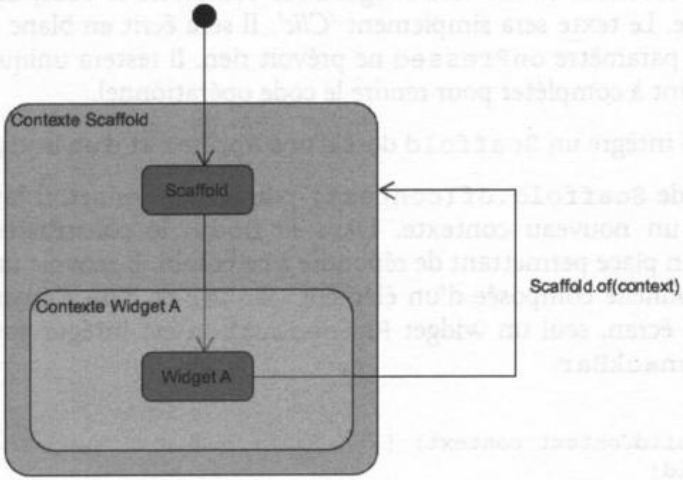


Figure 1 - Appel de Scaffold.of(context)

Pour mettre ceci en application, il est important de créer une instance de `SnackBar`. Celle-ci est donc placée comme variable au début de la classe `MyHomePage`.

```
class MyHomePage extends StatelessWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;
 final SnackBar _snack = SnackBar(
   content: Text('Ceci est une SnackBar !'),
   duration: Duration(seconds: 4),
   backgroundColor: Colors.red,
   action: SnackBarAction(
       label: 'Clic',
       textColor: Colors.white,
       onPressed: () {
         //Action à effectuer
       }
   ),
 );
...
}
```

Cette SnackBar comprendra un texte (*'Ceci est une SnackBar'*). Elle s'affichera pendant quatre secondes et elle sera rouge. Pour compléter le code, une action est ajoutée. Le texte sera simplement *'Clic'*. Il sera écrit en blanc et dans l'immédiat le paramètre onPressed ne prévoit rien. Il restera uniquement ce dernier point à compléter pour rendre le code opérationnel.

Le build principal intègre un Scaffold doté d'une appBar et d'un body.

Pour que la méthode Scaffold.of(context) puisse fonctionner, il faut maintenant créer un nouveau contexte. Dans le body, le constructeur Builder est mis en place permettant de répondre à ce besoin. Il renvoie une disposition traditionnelle composée d'un élément Center et d'un élément Column. Dans cet écran, seul un widget RaisedButton est intégré pour faire apparaître la snackBar.

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(title),
   ),
   body: Builder(builder: (BuildContext context) {
     return Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           RaisedButton(
             onPressed: () {
               Scaffold.of(context).showSnackBar(_snack);
             },
             color: Colors.blue,
             textColor: Colors.white,
             child: Text('Afficher la SnackBar'),
           ),
         ],
       ),
     );
   })
 );
}
```

© Editions ENI - All rights reserved

La méthode `showSnackBar` s'exécute dans le contexte du `Scaffold`. Elle prend comme paramètre l'intance `_snack` qui a été créée un peu plus haut. Le résultat obtenu alors est le suivant :

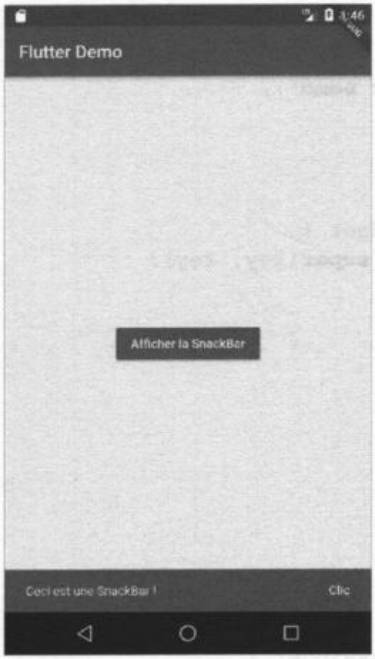


Figure 2 - SnackBar

Cette façon de procéder, qui consiste à créer un nouveau *build* via `Builder` n'est pas la meilleure. La documentation préconise de découper le programme en créant une nouvelle classe héritant de `StatelessWidget` qui va correspondre à l'élément `body` du `Scaffold`. Cet héritage va forcer la mise en place d'un nouveau *build*.

Le code va prendre cette forme :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo'),
    );
  }
}

class MyHomePage extends StatelessWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
       appBar: AppBar(
         title: Text(title),
       ),
       body: BodyDuScaffold());
 }
}

class BodyDuScaffold extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         RaisedButton(
             child: Text('Afficher la SnackBar encore'),
             color: Colors.blue,
             textColor: Colors.white,
             onPressed: () {
               final SnackBar _snack = SnackBar(
                 content: Text('Ceci est une SnackBar !'),
                 duration: Duration(seconds: 4),
                 backgroundColor: Colors.red,
                 action: SnackBarAction(
```

© Editions ENI - All rights reserved

```
                    label: 'Clic',
                    textColor: Colors.white,
                    onPressed: () {
                      //Action à effectuer
                    },
                  );
                  Scaffold.of(context).showSnackBar(_snack);
                },
              ],
            ),
          );
        }
      }
```

Le résultat est exactement le même tant visuellement que fonctionnellement. Le rendu est donc semblable à celui de la figure 2.

3. AlertDialog et CupertinoAlertDialog

Les widgets `AlertDialog` et `CupertinoAlertDialog` sont des pop-ups permettant d'afficher un message d'alerte à destination de l'utilisateur. Ils permettent de focaliser son attention.

Ils répondent à la nécessité d'obtenir une réponse de l'utilisateur ou de lui signifier un élément important.

Pour permettre leur affichage, une méthode nommée `showDialog` peut être appelée. Elle attend obligatoirement un paramètre définissant un contexte grâce à `BuildContext`. Celui-ci renverra l'objet `AlertDialog` ou `CupertinoAlertDialog`.

Les messages pouvant être juste informatifs et ne nécessitant pas toujours de réponse de l'utilisateur, un clic effectué en dehors du pop-up le fermera par défaut. Pour éviter ce comportement, notamment quand une action est attendue, le paramètre `barrierDismissible` peut prendre la valeur `false`. Dès lors, l'utilisateur sera forcé de passer par une des actions prévues et ne pourra plus fermer le dialogue en appuyant en dehors.

3.1 AlertDialog

Les boîtes de dialogue `AlertDialog` sont de style Material Design. Elles possèdent un constructeur qui attend un paramètre nommé `content` qui est, bien souvent, un texte, mais qui peut aussi être une icône ou une image.

Afin de pouvoir interagir avec l'utilisateur, le paramètre `action` peut être mis en place. Il attend une liste de widgets.

Pour le côté esthétique, il est possible de définir une valeur de `backgroundColor`, de choisir une `elevation` ou encore de modifier l'apparence du contour avec `shape`.

Voici le code d'une application dans laquelle un bouton va déclencher l'ouverture d'une `AlertDialog`. Ici, l'apparence est laissée de côté pour obtenir un rendu par défaut :

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter AlertDialog'),
   );
 }
}

class MyHomePage extends StatelessWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(title),
     ),
```

© Editions ENI - All rights reserved

```
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            RaisedButton(
                child: Text('Ouvrir une AlertDialog'),
                textColor: Colors.white,
                color: Colors.red,
                onPressed: () {
                  showDialog(
                      context: context,
                      barrierDismissible: false,
                      builder: (BuildContext context) {
                        return AlertDialog(
                          title: Text('Information'),
                          content: Text("Ceci est un message d'AlertDialog"),
                          actions: <Widget>[
                            FlatButton(
                                onPressed: () {
                                  Navigator.of(context).pop();
                                },
                                child: Text('Fermer'))
                          ],
                        );
                      });
                },
          ],
        ),
      ),
    );
  }
}
```

À l'appui sur le bouton rouge (`RaisedButton`), `onPressed` appelle la méthode `showDialog`. Pour forcer l'utilisateur à interagir avec la fenêtre d'information, la propriété `barrierDismissible` est passée à `false`.

Le `builder` retourne une instance d'`AlertDialog` qui possède un titre (`title`), un contenu (`content`) et un comportement (`actions`). Dans cette dernière propriété, afin de rester cohérent, un bouton au style Material Design est implanté. Le choix se porte sur un `FlatButton`. Le dialogue pop-up possédant déjà une élévation, il est préférable d'éviter d'en rajouter une nouvelle à l'intérieur avec un `RaisedButton` par exemple. Le `FlatButton` est un bouton "à plat", il se prête donc parfaitement à l'exercice.

Ce bouton appelle, via `onPressed`, la méthode `Navigator.of(context).pop()` qui sera détaillée un peu plus loin dans ce chapitre.

Voici à quoi ressemble cette `AlertDialog` :

Figure 3 - AlertDialog

© Editions ENI - All rights reserved

3.2 CupertinoAlertDialog

Les CupertinoAlertDialog ne sont qu'une copie des AlertDialog au style iOS. L'implémentation est identique. Dans le builder, il faut cette fois-ci intégrer une instance de CupertinoAlertDialog. Dans la propriété actions, le bouton de fermeture de la fenêtre ne sera plus un widget FlatButton, mais un CupertinoDialogAction qui va reprendre le style de la plateforme.

Pour tester cela, le code suivant est ajouté comme second enfant de la Column :

```
RaisedButton(
   child: Text('Ouvrir une CupertinoAlertDialog'),
   textColor: Colors.white,
   color: Colors.blue,
   onPressed: () {
     showDialog(
         context: context,
         barrierDismissible: false,
         builder: (BuildContext context) {
           return CupertinoAlertDialog(
             title: Text('Information'),
             content: Text(
                 "Ceci est un message de CupertinoAlertDialog"),
             actions: <Widget>[
               CupertinoDialogAction(
                 child: Text('Fermer'),
                 onPressed: () {
                   Navigator.of(context).pop();
                 },
               )
             ],
           );
         });
   }),
```

Un nouveau `RaisedButton` (bleu) est mis en place. C'est lui qui déclenchera l'ouverture du pop-up. Le rendu est assez différent du précédent :

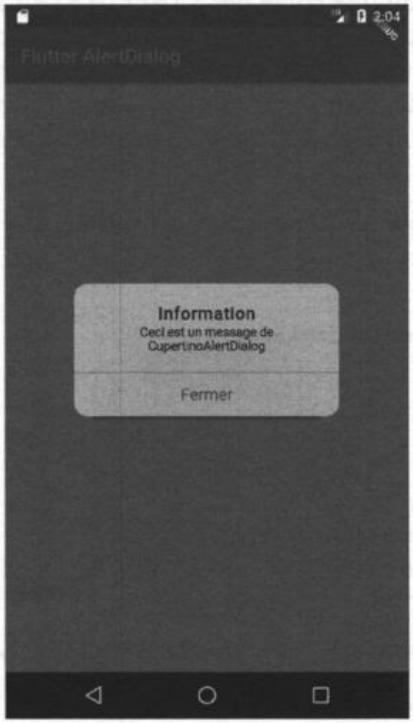


Figure 4 - CupertinoAlertDialog

4. SimpleDialog

Les `SimpleDialog` et `CupertinoDialog` sont des pop-ups utilisés dans le but d'offrir un choix à l'utilisateur ou de lui afficher une information.

Elles doivent être considérées comme ayant une priorité moindre que les `AlertDialog`.

© Editions ENI - All rights reserved

L'implémentation est réalisée de manière identique à celle d'`AlertDialog`. Il faudra donc faire appel à la méthode `showDialog`. Pour rappel, elle attend obligatoirement un paramètre définissant un contexte grâce à `BuildContext`. Celui-ci renverra l'`AlertDialog` ou la `CupertinoAlertDialog`.

Les messages pouvant être juste informatifs et ne nécessitant pas toujours de réponse de l'utilisateur ; par défaut, un clic en dehors du pop-up le fermera. Pour éviter ce comportement, notamment quand une action est attendue, le paramètre `barrierDismissible` pourra prendre la valeur `false`. Dès lors, l'utilisateur sera forcé de passer par une des actions prévues et ne pourra plus fermer le pop-up en appuyant en dehors.

4.1 SimpleDialog

La classe `SimpleDialog` est de style Material Design. Elle permet d'intégrer des `SimpleDialogAction` qui géreront chacune un événement `onPressed` et posséderont un enfant (`child`).

Les `SimpleDialog` s'intègrent bien dans le cadre d'un `switch` dans la mesure où, si un choix est proposé à l'utilisateur, il pourra être géré au cas par cas.

Pour illustrer ce propos, un exemple d'application dotée d'un `StatefulWidget` est mis en place. Ainsi, il sera possible de gérer l'état (`state`).

L'objectif est d'offrir la possibilité à l'utilisateur de choisir un mode de transport parmi la voiture, l'avion et le bateau. Ce choix va être effectué dans la `SimpleDialog` et sera confirmé sur l'écran principal dans un texte. Il faut donc que la valeur choisie soit récupérée et exploitée.

Une énumération est créée en dehors de la classe `_MyHomePageState`. Elle permet de gérer une liste que l'on pourrait qualifier de fixe :

```
enum ListeChoix {
 Voiture,
 Avion,
 Bateau
}
```

Dans la classe `_MyHomePageState`, une première variable est placée. Elle va servir à afficher le choix de l'utilisateur :

```
String _choix = 'Aucun choix';
```

Il reste maintenant à construire la partie visuelle de l'application. Dans le `build`, le code suivant est mis en place :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.green,
   ),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         RaisedButton(
           child: Text('Ouvrir une SimpleDialog'),
           color: Colors.green,
           textColor: Colors.white,
           onPressed: _quelChoix,
         ),
         Padding(padding: EdgeInsets.only(bottom: 30)),
         Text(
           'Voici le choix fait avec le SimpleDialog :',
         ),
         Padding(padding: EdgeInsets.only(bottom: 15)),
         Text(
           '$_choix',
           style: TextStyle(
             color: Colors.green,
             fontWeight: FontWeight.bold,
             fontSize: 40
           ),
         ),
       ],
     ),
   ),
   backgroundColor: Colors.green[50],
 );
}
```

© Editions ENI - All rights reserved

Dans le body, un RaisedButton va servir à déclencher l'ouverture de la SimpleDialog. Cette fois-ci, cette étape va être déportée dans une fonction nommée _quelChoix.

Deux textes sont placés ensuite. Ces trois éléments sont espacés par des widgets Padding qui n'ont aucun autre rôle ici que d'aérer l'écran.

Le premier texte est une phrase générique. Le second affiche le choix effectué par l'utilisateur dans la SimpleDialog. Pour cela, la variable _choix créée un peu plus haut est utilisée.

Les autres paramètres du code ne sont pas détaillés car leur fonction est seulement esthétique.

Il est temps d'examiner la fonction centrale de ce programme, _quelChoix :

```
Future<void> _quelChoix() async {
 switch (await showDialog<ListeChoix>(
     context: context,
     barrierDismissible: false,
     builder: (BuildContext context) {
       return SimpleDialog(
         title: const Text('Choisissez un mode de transport :'),
         children: <Widget>[
           SimpleDialogOption(
             child: Row(
               mainAxisAlignment: MainAxisAlignment.start,
               children: <Widget>[
                 Icon(Icons.directions_car),
                 const Text(' Voiture')
               ],
             ),
             onPressed: () {
               Navigator.pop(context, ListeChoix.Voiture);
               },
           ),
           SimpleDialogOption(
             child: Row(
               mainAxisAlignment: MainAxisAlignment.start,
               children: <Widget>[
                 Icon(Icons.airplanemode_active),
                 const Text(' Avion')
               ],
             ),
```

```
                onPressed: () {
                  Navigator.pop(context, ListeChoix.Avion);
                  },
              ),
              SimpleDialogOption(
                child: Row(
                  mainAxisAlignment: MainAxisAlignment.start,
                  children: <Widget>[
                    Icon(Icons.directions_boat),
                    const Text(' Bateau')
                  ],
                ),
                onPressed: () {
                  Navigator.pop(context, ListeChoix.Bateau);
                  },
              ),
            ],
          );
        }
    )){
      case ListeChoix.Voiture:
        _choixDuSimpleDialog('Voiture');
        break;
      case ListeChoix.Avion:
        _choixDuSimpleDialog('Avion');
        break;
      case ListeChoix.Bateau:
        _choixDuSimpleDialog('Bateau');
        break;
    }
  }
```

Cette fonction est une `Future` (cf. chapitre Dart, notions avancées, section Traitement asynchrone). Cela n'est pas indispensable, mais c'est une manière plus saine de procéder quand un élément est attendu. Ici, c'est le choix de l'utilisateur qu'il faut attendre.

© Editions ENI - All rights reserved

À l'intérieur, une structure switch est mise en place. Pour rappel, switch, dans une forme plus simplifiée, s'écrit comme suit :

```
switch(uneVariable) {
  case premiereValeurPossible:
     //Action qui découle de ce choix
     break ;
  case secondeValeurPossible:
     //Action qui découle de ce choix
     break ;
  ...
}
```

Dans l'application, il s'agit de la même chose. La variable uneVariable correspond à l'appel à la méthode showDialog<ListeChoix> qui va elle-même servir à ouvrir la SimpleDialog.

Trois SimpleDialogOption sont intégrées dans le pop-up. Chacune d'elles est une ligne (Row) composée d'une icône et d'un texte. Elles ont aussi à gérer un événement onPressed qui va appeler la méthode Navigator.pop(contexteCourant, résultat). Cette notion sera détaillée un peu plus loin dans ce chapitre. Ce qu'il faut comprendre, c'est que ce choix va ainsi remonter. La valeur retournée sera une des valeurs de l'énumération :

- ListeChoix.Voiture
- ListeChoix.Avion
- ListeChoix.Bateau

Finalement, dans la structure switch, maVariable va prendre une de ces trois valeurs. Enfin, les alternatives case sont mises en place avec les valeurs de l'énumération. Pour chaque cas, une méthode nommée _choixDuSimpleDialog, comportant un appel à setState, va être invoquée. Elle va mettre à jour la valeur de _choix pour que l'écran affiche le mode de transport choisi :

```
void _choixDuSimpleDialog(String choix) {
 setState(() {
   _choix = choix;
 });
}
```

Avant d'observer le rendu, étant donné que le code de cette application est un peu complexe, le voici complet afin que personne ne s'y perde :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo SimpleDialog'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
 String _choix = 'Aucun choix';

 void _choixDuSimpleDialog(String choix) {
   setState(() {
     _choix = choix;
   });
 }

 Future<void> _quelChoix() async {
   switch (await showDialog<ListeChoix>(
       context: context,
       barrierDismissible: false,
       builder: (BuildContext context) {
         return SimpleDialog(
```

© Editions ENI - All rights reserved

```
        title: const Text('Choisissez un mode de transport :'),
        children: <Widget>[
          SimpleDialogOption(
            child: Row(
              mainAxisAlignment: MainAxisAlignment.start,
              children: <Widget>[
                Icon(Icons.directions_car),
                const Text(' Voiture')
              ],
            ),
            onPressed: () {
              Navigator.pop(context, ListeChoix.Voiture);
            },
          ),
          SimpleDialogOption(
            child: Row(
              mainAxisAlignment: MainAxisAlignment.start,
              children: <Widget>[
                Icon(Icons.airplanemode_active),
                const Text(' Avion')
              ],
            ),
            onPressed: () {
              Navigator.pop(context, ListeChoix.Avion);
            },
          ),
          SimpleDialogOption(
            child: Row(
              mainAxisAlignment: MainAxisAlignment.start,
              children: <Widget>[
                Icon(Icons.directions_boat),
                const Text(' Bateau')
              ],
            ),
            onPressed: () {
              Navigator.pop(context, ListeChoix.Bateau);
            },
          ),
        ],
      );
    }
  )){
    case ListeChoix.Voiture:
      _choixDuSimpleDialog('Voiture');
```

```
      break;
    case ListeChoix.Avion:
      _choixDuSimpleDialog('Avion');
      break;
    case ListeChoix.Bateau:
      _choixDuSimpleDialog('Bateau');
      break;
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
      backgroundColor: Colors.green,
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          RaisedButton(
              child: Text('Ouvrir une SimpleDialog'),
              color: Colors.green,
              textColor: Colors.white,
              onPressed: _quelChoix,
          ),
          Padding(padding: EdgeInsets.only(bottom: 30)),
          Text(
            'Voici le choix fait avec le SimpleDialog :',
          ),
          Padding(padding: EdgeInsets.only(bottom: 15)),
          Text(
            '$_choix',
            style: TextStyle(
              color: Colors.green,
              fontWeight: FontWeight.bold,
              fontSize: 40
            ),
          ),
        ],
      ),
    ),
    backgroundColor: Colors.green[50],
```

© Editions ENI - All rights reserved

```
  );
  }
}


enum ListeChoix {
 Voiture,
 Avion,
 Bateau
}
```

Après avoir appuyé sur le bouton, l'écran affiche la SimpleDialog :

Figure 5 - SimpleDialog

Pour l'exemple, imaginons que l'utilisateur choisisse l'avion comme mode de transport. L'écran principal va donc répéter ce choix :

Figure 6 - Récupération du choix du SimpleDialog

4.2 CupertinoDialog

Il existe un équivalent à `SimpleDialog` pour le style iOS. Il s'agit de `CupertinoDialog`. Toutefois, ce widget a été déprécié à partir de la version 0.2.3 de Flutter.

Il est plutôt conseillé d'utiliser `CupertinoAlertDialog` à la place.

© Editions ENI - All rights reserved

5. Feuilles de choix

Dans le même style que ce qui a été vu juste au-dessus, il existe ce qui pourrait être traduit en français par des feuilles de choix, plus explicitement des `BottomSheet` et des `CupertinoActionSheet`. Elles correspondent respectivement aux styles Material Design et iOS. Elles peuvent être utilisées pour cacher du contenu un peu trop encombrant sur un écran, comme des liens de navigation.

5.1 BottomSheet

Il existe plusieurs moyens d'afficher une `BottomSheet`. Ce widget s'implémente rarement tel quel. Le plus souvent, il convient de passer par d'autres widgets. La documentation préconise que ce choix se portera soit sur `ScaffoldState.showBottomSheet` ou sur `showModalBottomSheet`. Dans l'exemple qui suit, seul le second va être développé.

En s'appuyant sur le code de l'application précédente où `SimpleDialog` était paramétré pour fonctionner avec une énumération, un nouveau bouton déclencheur va être intégré juste en-dessous du premier :

```
RaisedButton(
 child: Text('Ouvrir une BottomSheet'),
 color: Colors.green[700],
 textColor: Colors.white,
 onPressed: _quelChoixBottom,
),
```

Cette fois-ci, la fonction appelée est `_quelChoixBottom`. Voici d'abord le code complet de la fonction, détaillée juste après :

```
Future<void> _quelChoixBottom() async {
 switch (await showModalBottomSheet<ListeChoix>(
     context: context,
     isDismissible: false,
     builder: (BuildContext context) {
       return Center(
         child: Container(
           color: Colors.green[200],
           child: Column(
             mainAxisAlignment: MainAxisAlignment.spaceAround,
```

```
children: <Widget>[
  Text(
    'Choisissez un mode de transport :',
    style: TextStyle(
      fontWeight: FontWeight.bold,
      fontSize: 25,
    ),
    textAlign: TextAlign.center,
  ),
  Divider(),
  SimpleDialogOption(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.start,
      children: <Widget>[
        Icon(Icons.directions_car),
        const Text(' Voiture')
      ],
    ),
    onPressed: () {
      Navigator.pop(context, ListeChoix.Voiture);
    },
  ),
  Divider(),
  SimpleDialogOption(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.start,
      children: <Widget>[
        Icon(Icons.airplanemode_active),
        const Text(' Avion')
      ],
    ),
    onPressed: () {
      Navigator.pop(context, ListeChoix.Avion);
    },
  ),
  Divider(),
  SimpleDialogOption(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.start,
      children: <Widget>[
        Icon(Icons.directions_boat),
        const Text(' Bateau')
      ],
    ),
```

© Editions ENI - All rights reserved

```
                onPressed: () {
                  Navigator.pop(context, ListeChoix.Bateau);
                },
              ),
            ],
          ),
        )
      );
    }
  )){
    case ListeChoix.Voiture:
      _choixDuSimpleDialog('Voiture',2);
      break;
    case ListeChoix.Avion:
      _choixDuSimpleDialog('Avion',2);
      break;
    case ListeChoix.Bateau:
      _choixDuSimpleDialog('Bateau',2);
      break;
  }
}
```

La structure globale est très semblable à celle vue précédemment. Ce choix est volontaire. Il permet de simplifier l'approche vis-à-vis des `BottomSheet`.

La première différence notable réside dans `switch`, qui prend cette fois-ci comme variable la classe `showModalBottomSheet`.

Remarque

À noter que, pour éviter que l'utilisateur puisse fermer la fenêtre en cliquant en dehors, le paramètre a changé. Il s'agit de `isDismissible` et non plus de `barrierDismissible`.

Quant au reste, il y a peu de changement. Des `Divider` sont insérés pour faciliter la lecture des éléments. Dans la mesure où il n'y a pas de titre proposé, un texte a été inséré pour jouer ce rôle.

Enfin, pour permettre de bien identifier l'origine du choix entre la SimpleDialog et la BottomSheet, une modification a été apportée. La phrase 'Voici le choix fait avec la SimpleDialog' se transforme en :

```
Text(
 'Voici le choix fait avec $_source :',
),
```

_source est déclarée en début de classe de cette manière :

```
String _source = '';
```

Les cas prévus restent également presque les mêmes et font toujours appel à _choixSimpleDialog. Simplement, un paramètre supplémentaire est ajouté. Il va servir à préciser la source à l'origine du choix. Pour la SimpleDialog, le nombre 1 sera passé. Tandis que pour la BottomSheet, ce sera le 2.

La méthode _choixSimpleDialog n'est pas renommée, mais corrigée pour accepter ce nouveau paramètre et le traiter dans un switch/case qui fera évoluer la variable _source :

```
void _choixDuSimpleDialog(String choix, int source) {
  setState(() {
    switch(source) {
      case 1:
        _source = 'SimpleDialog';
        break;
      case 2:
        _source = 'BottomSheet';
        break;
    }
    _choix = choix;
  });
}
```

© Editions ENI - All rights reserved

Le widget `BottomSheet` prend cette apparence :

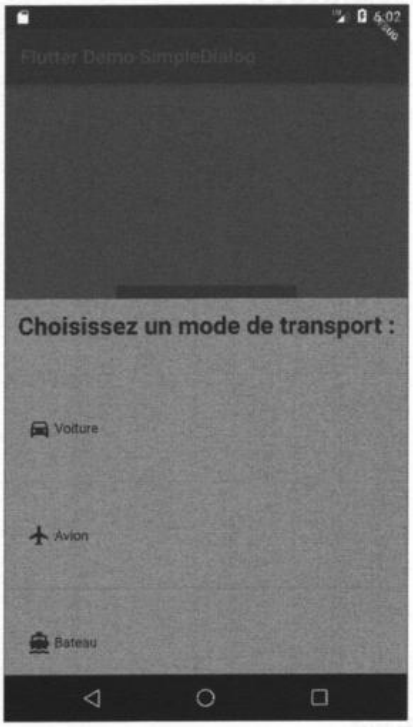


Figure 7 - BottomSheet

Une fois le choix effectué, la page principale s'adapte et l'affiche. Elle indique également que ce dernier provient de `BottomSheet` :

Figure 8 - Écran principal relevant le choix effectué

5.2 CupertinoActionSheet

Dans la même lignée que `BottomSheet`, il existe un widget au style iOS. Il s'agit de `CupertinoActionSheet`. Le principe reste toujours le même.

Pour commencer, il faut un nouveau bouton déclencheur :

```
RaisedButton(
 child: Text('Ouvrir une CupertinoActionSheet'),
 color: Colors.green[800],
 textColor: Colors.white,
 onPressed: _quelChoixCupertino,
),
```

© Editions ENI - All rights reserved

Ce bouton est placé juste après celui destiné à BottomSheet. Une fonction nommée _quelChoixCupertino doit être intégrée. C'est elle qui sera appelée via onPressed.

```
Future<void> _quelChoixCupertino() async {
 switch (await showCupertinoModalPopup<ListeChoix>(
     context: context,
     builder: (BuildContext context) {
       return CupertinoActionSheet(
         title: const Text('Choisissez un mode de transport :'),
         actions: <Widget>[
           CupertinoActionSheetAction(
             child: Row(
               mainAxisAlignment: MainAxisAlignment.start,
               children: <Widget>[
                 Icon(Icons.directions_car),
                 const Text(' Voiture')
               ],
             ),
             onPressed: () {
               Navigator.pop(context, ListeChoix.Voiture);
             },
           ),
           CupertinoActionSheetAction(
             child: Row(
               mainAxisAlignment: MainAxisAlignment.start,
               children: <Widget>[
                 Icon(Icons.airplanemode_active),
                 const Text(' Avion')
               ],
             ),
             onPressed: () {
               Navigator.pop(context, ListeChoix.Avion);
             },
           ),
           CupertinoActionSheetAction(
             child: Row(
               mainAxisAlignment: MainAxisAlignment.start,
               children: <Widget>[
                 Icon(Icons.directions_boat),
                 const Text(' Bateau')
               ],
             ),
             onPressed: () {
```

```
                Navigator.pop(context, ListeChoix.Bateau);
              },
            ),
          ],
        );
      }
  )){
    case ListeChoix.Voiture:
      _choixDuSimpleDialog('Voiture',3);
      break;
    case ListeChoix.Avion:
      _choixDuSimpleDialog('Avion',3);
      break;
    case ListeChoix.Bateau:
      _choixDuSimpleDialog('Bateau',3);
      break;
  }
}
```

Encore une fois, la structure reste très proche de l'exemple précédent. Toutefois, l'ouverture de la `CupertinoActionSheet` intervient avec la méthode `showCupertinoModalPopup`.

Pour les `actions`, il existe le widget `CupertinoActionSheetActions`.

Il s'agit toujours d'un `switch/case`. Les cas continuent d'appeler `_choixDuSimpleDialog`. Mais le nombre 3 est donné en guise de valeur pour `source`.

Il convient enfin de modifier cette méthode pour qu'elle prenne en considération cette troisième possibilité :

```
void _choixDuSimpleDialog(String choix, int source) {
 setState(() {
   switch(source) {
     case 1:
       _source = 'SimpleDialog';
       break;
     case 2:
       _source = 'BottomSheet';
       break;
     case 3:
       _source = 'CupertinoActionSheet';
       break;
```

© Editions ENI - All rights reserved

```
    }
    _choix = choix;
  });
}
```

Tout est complet, il ne reste plus qu'à sauvegarder ces ajouts et modifications pour que le *hot reload* nous permette d'observer l'apparence de `CupertinoActionSheet` après avoir appuyé sur le bouton :

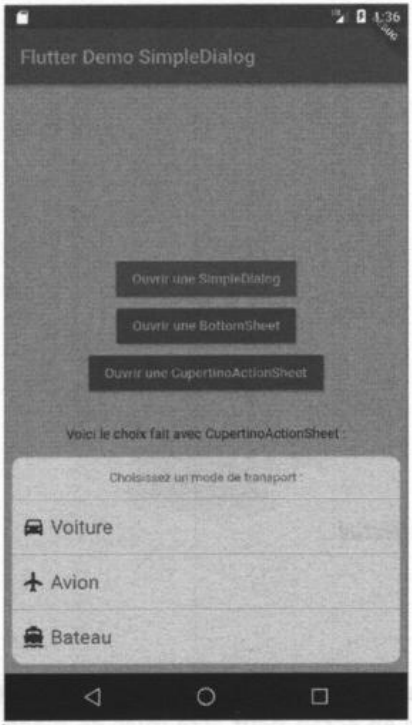


Figure 9 - CupertinoActionSheet

Une fois le choix effectué, l'écran d'accueil se met à jour. Dans l'exemple, c'est le transport en bateau qui a été privilégié :

Figure 10 - Écran principal relevant le choix effectué

6. Routes

Il est maintenant temps de s'intéresser à un point crucial, les routes. L'objectif est de voir comment la navigation d'écran en écran peut être mise en place.

Le widget chargé de ce rôle est la classe `Navigator`. Il fonctionne en plaçant les écrans les uns devant les autres sous la forme d'une pile. Cet effet de superposition permet en outre de revenir en arrière puisque cette pile reste mémorisée.

Pour être vraiment efficaces, les routes peuvent être définies à l'avance sous une forme proche d'une URL.

© Editions ENI - All rights reserved

6.1 Navigator.push

Pour passer à un autre écran, il suffit d'utiliser `Navigator.push`. Deux paramètres sont attendus. Le premier est le contexte courant. Il permettra de revenir en arrière au moment opportun. Le second est un objet `MaterialPageRoute`. Un nouveau contexte va être construit ici et indiquer vers quelle page se diriger.

Pour bien comprendre cette mise en place, voici un premier écran comportant un bouton (Figure 11) :

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('$title'),
      backgroundColor: Colors.teal,
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            "Ecran n°1",
            style: TextStyle(fontWeight: FontWeight.bold,
fontSize: 40, color: Colors.teal),
          ),
          Padding(padding: EdgeInsets.only(bottom: 20)),
          Text("Appuyer sur ce bouton pour passer à l'écran suivant"),
          Padding(padding: EdgeInsets.only(bottom: 20)),
          RaisedButton(
              child: Text('Suivant'),
              color: Colors.teal,
              textColor: Colors.white,
              onPressed: null
          ),
        ],
      ),
    ),
  );
}
```

Maintenant que la partie graphique est en place, il ne reste plus qu'à définir le comportement suivant l'appui sur ce bouton. Dans onPressed, le code suivant va être ajouté :

```
onPressed: () {
 Navigator.push(
     context,
     MaterialPageRoute(
         builder: (BuildContext context) =>
             MySecondPage(title: title)));
 }
```

La méthode Navigator.push est mise en place. Comme prévu le contexte courant est passé en paramètre ainsi qu'un objet MaterialPageRoute. Ce dernier attend un builder. Un nouveau contexte est créé, il retourne une instance de MySecondPage grâce au constructeur.

Cette seconde page est décrite comme ceci (Figure 11) :

```
class MySecondPage extends StatelessWidget {
 MySecondPage({this.title});

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('$title'),
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             "Ecran n°2",
             style: TextStyle(fontWeight: FontWeight.bold,
fontSize: 40, color: Colors.blue),
           ),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           ],
       ),
     ),
   );
 }
}
```

© Editions ENI - All rights reserved

Pour l'instant, cet écran est simple, mais il permet de vérifier que la route mise en place fonctionne correctement.

Il faut maintenant prévoir le retour en arrière.

6.2 Navigator.pop

Avec `Navigator.push`, une pile de routes est mise en place. La fonction `pop` de `Navigator` permet de revenir à la route précédente. Elle permet donc de descendre d'une couche. Au moment de son exécution, elle supprime la route courante de la pile.

Dans la classe `MySecondPage`, il faut ajouter un bouton déclenchant ce retour arrière. En dessous du `Padding`, ce code va être inséré :

```
Text("Appuyer sur ce bouton pour revenir à l'écran précédent"),
Padding(padding: EdgeInsets.only(bottom: 20)),
RaisedButton(
   child: Text('Précédent'),
   color: Colors.blue,
   textColor: Colors.white,
   onPressed: () {
     Navigator.pop(context);
   }),
```

Dans `onPressed` du `RaisedButton`, la méthode `Navigator.pop(context)` est appelée.

En termes de rendu, ceci est obtenu :

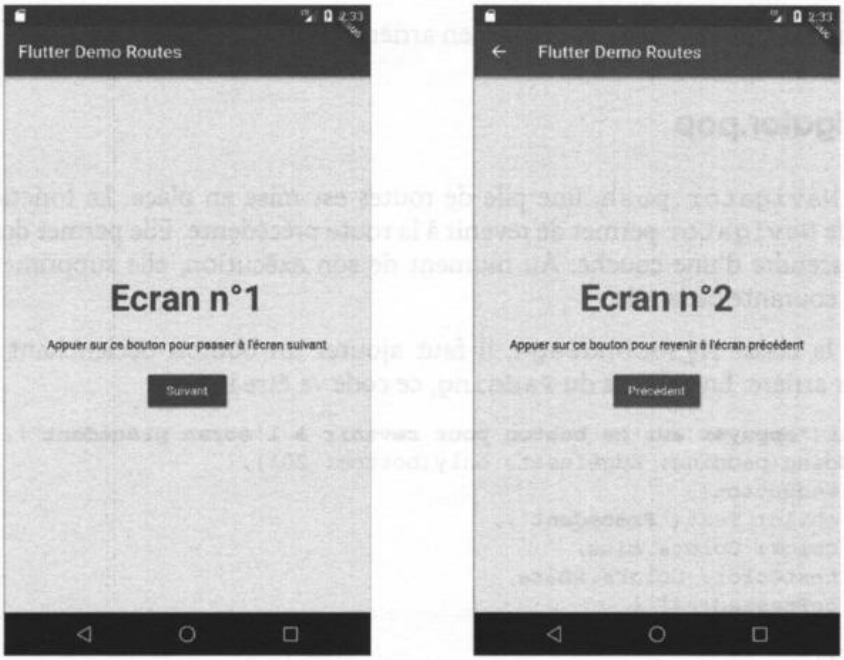


Figure 11 - Navigator.push

6.3 Navigator.pushNamed

Pour faciliter la maintenance de l'application, il est aussi possible de référencer les différentes routes qui serviront tout au long du projet.

Dans la classe qui sert au lancement (`runApp`), il existe un paramètre du constructeur `MaterialApp` qui se nomme `routes`. Il va accueillir une *map* possédant une clé de type chaîne de caractères et une valeur de type `WidgetBuilder`.

© Editions ENI - All rights reserved

Dans un nouveau projet, trois pages sont créées : `MyFirstPage`, `MySecondPage` et `MyThirdPage`. Le code qui permettra le référencement des routes pourrait être le suivant :

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyFirstPage(title: 'Flutter Demo Home Page'),
     routes: <String, WidgetBuilder> {
       '/route1': (BuildContext context) => MyFirstPage(title: 'Page 1'),
       '/route2': (BuildContext context) => MySecondPage(title: 'Page 2'),
       '/route3': (BuildContext context) => MyThirdPage(title: 'Page 3'),
     },
   );
 }
}
```

Au niveau des clés, des chaînes de caractères faisant penser à des URL sont placées. En guise de valeurs, des fonctions qui renvoient vers les constructeurs des différentes classes correspondants aux trois pages sont affectées.

Une fois référencées de la sorte, les pages peuvent être appelées grâce à la méthode `Navigator.pushNamed` qui prend deux paramètres : le contexte et une des clés définies dans `routes`. Par exemple, en ce qui concerne l'appel de la seconde page :

```
Navigator.pushNamed(context, '/route2');
```

L'avantage procuré par cette façon de procéder est qu'elle permet de placer à un seul endroit les références aux routes. Il y a une dépendance moins forte entre les classes (absence d'importation notamment). Enfin, la maintenance s'en trouve donc facilitée.

Pour tester cela, voici le détail des trois classes servant dans l'application. Il s'agit de trois classes très similaires :

- MyFirstPage :

```
import 'package:flutter/material.dart';

class MyFirstPage extends StatelessWidget {
 MyFirstPage({this.title});

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('$title'),
       backgroundColor: Colors.green,
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             "Ecran n°1",
             style: TextStyle(
               fontWeight: FontWeight.bold,
               fontSize: 40,
               color: Colors.green),
           ),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           Text("Appuyer sur un bouton pour passer à un autre écran"),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           RaisedButton(
               child: Text('Ecran n°2'),
               color: Colors.teal,
               textColor: Colors.white,
               onPressed: () {
                 Navigator.pushNamed(context, '/route2');
               }),
           RaisedButton(
               child: Text('Ecran n°3'),
               color: Colors.blue,
               textColor: Colors.white,
               onPressed: () {
                 Navigator.pushNamed(context, '/route3');
               }),
```

© Editions ENI - All rights reserved

```
        ],
      ),
    ),
    backgroundColor: Colors.green[100],
  );
 }
}
```

– MySecondPage :

```
import 'package:flutter/material.dart';

class MySecondPage extends StatelessWidget {
 MySecondPage({this.title});

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('$title'),
       backgroundColor: Colors.teal,
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             "Ecran n°2",
            style: TextStyle(fontWeight: FontWeight.bold, fontSize: 40,
color: Colors.teal),
           ),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           Text("Appuyer sur ce bouton pour passer à l'écran suivant"),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           RaisedButton(
               child: Text('Ecran n°1'),
               color: Colors.green,
               textColor: Colors.white,
               onPressed: () {
                 Navigator.pushNamed(context, '/route1');
               }),
           RaisedButton(
               child: Text('Ecran n°3'),
               color: Colors.blue,
               textColor: Colors.white,
               onPressed: () {
```

```
                Navigator.pushNamed(context, '/route3');
              }),
          ],
        ),
      ),
      backgroundColor: Colors.teal[100],
    );
  }

}
```

- MyThirdPage :

```
import 'package:flutter/material.dart';

class MyThirdPage extends StatelessWidget {
 MyThirdPage({this.title});

 final String title;

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('$title'),
       backgroundColor: Colors.blue,
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text(
             "Ecran n°3",
             style: TextStyle(fontWeight: FontWeight.bold, fontSize: 40,
color: Colors.blue),
           ),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           Text("Appuyer sur ce bouton pour passer à l'écran suivant"),
           Padding(padding: EdgeInsets.only(bottom: 20)),
           RaisedButton(
               child: Text('Ecran n°1'),
               color: Colors.green,
               textColor: Colors.white,
               onPressed: () {
                 Navigator.pushNamed(context, '/route1');
               },
           RaisedButton(
               child: Text('Ecran n°2'),
               color: Colors.teal,
```

© Editions ENI - All rights reserved

```
                textColor: Colors.white,
                onPressed: () {
                  Navigator.pushNamed(context, '/route2');
                }),
            ],
          ),
        ),
        backgroundColor: Colors.blue[100],
      );
    }

  }
```

Sur chaque écran, deux boutons sont placés. Ils permettent de se diriger vers une des deux autres pages. Il est intéressant de constater l'effet de pile grâce à l'`AppBar`. Passé le premier écran, l'`AppBar` dispose d'une flèche de retour en arrière qui permet de remonter la pile comme le ferait la méthode `pop` de `Navigator`.

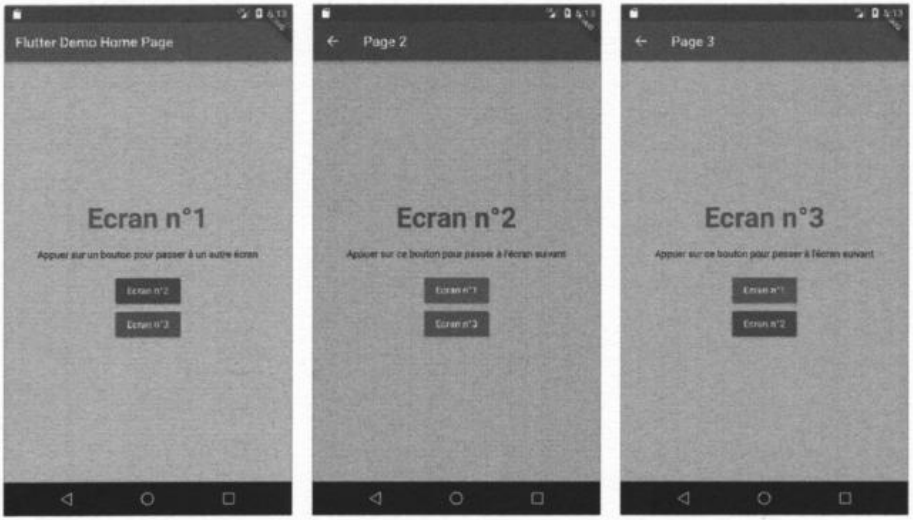
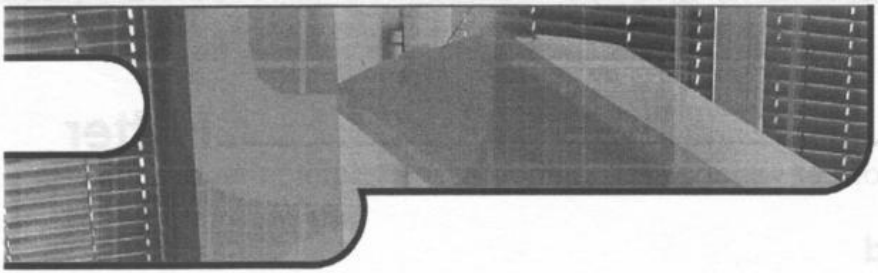


Figure 12 - Routes



Chapitre 9
Formulaires

1. Introduction

Dans la liste conséquente des widgets de Flutter, il en existe un certain nombre dédiés aux interactions avec les utilisateurs. Ces widgets actifs existent pour récupérer de l'information. Pour le Web, ils composent un groupe alimentant les formulaires. Dans ce chapitre, ces widgets vont être détaillés. Pour débuter, les champs qui permettent la saisie de texte seront étudiés. Le chapitre s'attardera ensuite sur les dates et heures. Enfin, les widgets permettant d'effectuer un choix comme les switches, les radio boutons, les checkbox ou encore les sliders seront abordés.

2. Les champs de saisie de texte

Les champs de saisie sont l'expression la plus simple et généraliste de ces widgets. Dans un espace dédié, l'utilisateur peut saisir du texte. On en retrouve pour toute sorte d'usage et notamment au moment de la connexion pour la récupération d'un login et d'un mot de passe. Il existe deux widgets qui peuvent jouer ce rôle : `TextField` et `CupertinoTextField`.

2.1 TextField

Le widget `TextField` est donc un champ de saisie d'un texte pour le style Material Design. Comme souvent avec les widgets, il possède de nombreuses propriétés permettant de modifier son apparence. Le paramètre `decoration` du constructeur est par exemple dédié à cette tâche. Il faut y associer une instance de type `InputDecoration`. Le constructeur de cette classe comprend une grande quantité de paramètres optionnels allant d'un texte de remplacement (`hintText`) à la mise en place d'une icône ou encore d'un label (`labelText`). Les couleurs des différents éléments sont modifiables également.

Pour comprendre son fonctionnement, une nouvelle application est créée. La classe `MyHomePage` étend `StatefulWidget` afin de pouvoir modifier l'affichage en fonction des saisies réalisées dans le champ `TextField`.

```
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo Champs texte'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}
```

© Editions ENI - All rights reserved

```
class _MyHomePageState extends State<MyHomePage> {
    //Code à insérer
  }
```

L'objectif est de placer un champ dans lequel l'utilisateur va pouvoir saisir un texte. Puis, dans une zone dédiée située plus bas, de récupérer cette saisie en deux temps. D'abord pendant la phase d'écriture puis à la validation.

Dans la classe `_MyHomePageState`, un `build` est mis en place. Il comporte comme à son habitude un `Scaffold` et une `AppBar`. Pour le body, cette fois il convient de prendre en considération le fait que quand l'utilisateur ouvrira son clavier, la zone visible de l'écran va réduire. Avec un couple classique de widgets `Center` et `Column`, il risque d'y avoir un problème. Avec cette façon de procéder, les zones non visibles vont créer un dépassement qui va générer une erreur de ce type :

```
═══ Exception caught by rendering library═══════════════
The following assertion was thrown during layout:
A RenderFlex overflowed by 253 pixels on the bottom.
```

Sur l'émulateur, l'erreur est visible également et affiche une barre comme celle-ci :

Figure 1 - Bandeau d'erreur sur l'émulateur en cas de dépassement

Pour remédier à cela, un nouveau widget va être utilisé. Il s'agit d'un `SingleChildScrollView`. Son rôle est de mettre en place un *scroll*, c'est-à-dire un défilement, si la zone devient trop grande.

À l'intérieur de ce dernier, `child` va accepter la combinaison `Center` et `Column`.

Pour rendre l'affichage du champ plus propre, un `Container` va également être intégré. C'est l'occasion d'introduire une autre notion : `MediaQuery.of(context)`. Voici le code qui sera détaillé juste après :

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    resizeToAvoidBottomPadding: true,
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: SingleChildScrollView(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Formulaire',
              style: TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: 40,
                color: Colors.blue
              ),
            ),
            Divider(),
            Padding(padding: EdgeInsets.only(bottom: 20)),
            Container(
              height: MediaQuery.of(context).size.height*0.7,
              width: MediaQuery.of(context).size.width*0.9,
              child: ...
            ),
          ],
        ),
      ),
    ),
  );
}
```

© Editions ENI - All rights reserved

Sur la propriété `MediaQuery.of(context)`, il est possible de récupérer la taille (hauteur et largeur) de l'écran avec `size`. Ensuite, cette dernière étant composée d'une hauteur (`height`) et d'une largeur (`width`), il n'est pas compliqué de récupérer l'une ou l'autre. À partir de là, rien n'empêche de leur appliquer un coefficient comme dans le code ci-dessus, respectivement 0,7 et 0,9.

Il va maintenant être possible d'insérer la zone `TextField`. De manière à montrer les effets de la propriété `decoration`, un texte de remplacement (`hintText`) est placé. Puis les deux méthodes `onChanged` et `onSubmitted` sont déclarées. Chacune possédera son `setState`. Ce dernier doit faire évoluer une valeur, plus précisément une chaîne de caractères qui doit être déclarée au début de `_MyHomePageState` :

```
class _MyHomePageState extends State<MyHomePage> {

 String _tf = 'Aucun texte saisi';
 String _tfS = 'Aucun texte soumis';

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     resizeToAvoidBottomPadding: true,
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: SingleChildScrollView(
       child: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: <Widget>[
             Text(
               'Formulaire',
               style: TextStyle(
                   fontWeight: FontWeight.bold,
                   fontSize: 40,
                   color: Colors.blue
               ),
             ),
             Divider(),
             Padding(padding: EdgeInsets.only(bottom: 20)),
             Container(
               height: MediaQuery.of(context).size.height*0.7,
               width: MediaQuery.of(context).size.width*0.9,
               child: Column(
```

```
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Text('1- Saisie d\'un texte dans une TextField :'),
                  TextField(
                    decoration: InputDecoration(
                      hintText: 'Saisissez ce que vous voulez !',
                    ),
                    onChanged: (value) {
                      setState(() {
                        _tf = value;
                      });
                    },
                    onSubmitted: (value) {
                      setState(() {
                        _tfS = value;
                      });
                    },
                  ),
                  Padding(padding: EdgeInsets.only(bottom: 40)),
                ],
              ),
            ),
          ],
        ),
      ),
    ),
  );
}
}
```

Pour pouvoir vérifier si les méthodes `onChanged` et `onSubmitted` sont correctement mises en place, il faut afficher les variables créées un peu plus haut. Cet affichage va prendre place dans un nouveau `Container`.

Pour un peu de fantaisie, il va être décoré grâce au widget `BoxDecoration` avec une couleur de fond bleue et une ombre (`boxShadow`) afin de donner un effet d'élévation.

Voici donc le code qui vient s'ajouter en dessous du dernier `Padding` :

```
Container(
  width: MediaQuery.of(context).size.width*0.9,
  //color: Colors.blue,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: <Widget>[
```

© Editions ENI - All rights reserved

```
      Padding(padding: EdgeInsets.only(top: 10)),
      Text(
        '1- Texte en cours de saisie dans TextField : ',
        style: _textStyleDefault,
      ),
      Text(
        '$_tf',
        style: _textStyleSaisie,
      ),
      Text(
        '1- Texte soumis dans TextField : ',
        style: _textStyleDefault,
      ),
      Text(
        '$_tfS',
        style: _textStyleSaisie,
      ),
      Padding(padding: EdgeInsets.only(bottom: 10)),
    ],
  ),
  decoration: BoxDecoration(
    color: Colors.blue,
    boxShadow: [BoxShadow(
      color: Colors.black,
      blurRadius: 25.0,
      spreadRadius: 5.0,
      offset: Offset(0, 15)
    )]
  ),
)
```

Pour compléter le style, les textes des titres seront blancs. Quant aux valeurs, elles seront inscrites en orange et en gras. Dans la mesure où cette opération va être répétitive, deux variables de type `TextStyle` sont déclarées en début de classe :

```
TextStyle _textStyleDefault = TextStyle(
 color: Colors.white,
);

TextStyle _textStyleSaisie = TextStyle(
 color: Colors.amber,
 fontWeight: FontWeight.bold,
);
```

Voici le rendu de base de l'application :

Figure 2 - TextField

© Editions ENI - All rights reserved

Voici l'évolution de l'écran au cours de la saisie :

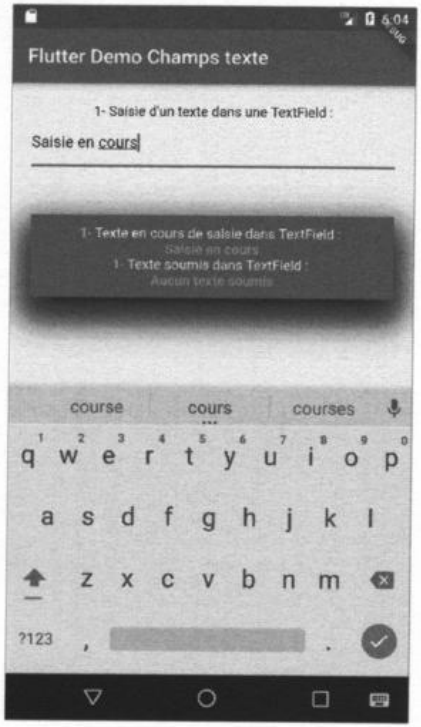


Figure 3 - TextField en cours de saisie

Et enfin, une fois que la saisie est validée et soumise :

Figure 4 - Saisie de la TextField validée

2.2 CupertinoTextField

`CupertinoTextField` fonctionne sur le même principe. Une différence est à relever toutefois, au niveau de la propriété `decoration` qui attend une instance de type `BoxDecoration`. Elle permet de modifier de manière concrète l'aspect du champ. Pour le texte de remplacement, la classe `BoxDecoration` ne le prévoyant pas, il existe une propriété `placeholder` pour jouer ce rôle.

© Editions ENI - All rights reserved

Dans l'exemple précédent, le champ `CupertinoTextField` est ajouté à la suite du champ `TextField`. Les deux éléments sont séparés par un `Padding` pour aérer l'ensemble :

```
Padding(padding: EdgeInsets.only(bottom: 10)),
Text('2- Saisie d\'un texte dans une CupertinoTextField :'),
CupertinoTextField(
 placeholder: 'Saisissez ce que vous voulez !',
 onChanged: (value) {
   setState(() {
     _cTf = value;
   });
 },
 onSubmitted: (value) {
   setState(() {
     _cTfS = value;
   });
 },
),
```

La méthodologie est la même. Les événements vont mettre à jour des variables correspondant aux chaînes de caractères. Celles-ci doivent être déclarées en début de classe :

```
String _cTf = 'Aucun texte saisi';
String _cTfS = 'Aucun texte soumis';
```

Il reste maintenant à mettre à jour la zone d'affichage en bas de l'écran. Elle se chargera de montrer les valeurs de `_cTf` et `_CtfS` :

```
Text(
 '2- Texte en cours de saisie dans CupertinoTextField : ',
 style: _textStyleDefault,
),
Text(
 '$_cTf',
 style: _textStyleSaisie,
),
Text(
 '2- Texte soumis dans CupertinoTextField : ',
 style: _textStyleDefault,
),
Text(
 '$_cTfS',
```

```
style: _textStyleSaisie,
),
```

Le rendu de l'application évolue et le champ `CupertinoTextField` apparaît comme ceci :

Figure 5 - CupertinoTextField

© Editions ENI - All rights reserved

2.3 Types de données

En fonction du type des données attendues dans le champ, il est possible de personnaliser le clavier ou de cacher la saisie.

2.3.1 Mot de passe

Pour améliorer la sécurité de l'application, il est d'usage, comme en programmation web, de cacher à la vue des curieux la saisie d'un mot de passe.

Les widgets `TextField` et `CupertinoTextField` possèdent pour cela un paramètre de type booléen `obscureText`. Si sa valeur est passée à `true`, le texte est caché et remplacé par des points.

2.3.2 Autres données

Pour des données précises, il est conseillé d'utiliser le paramètre `keyboardType`. Ce dernier permet de configurer le clavier en fonction du besoin et ainsi d'améliorer l'expérience utilisateur. Il attend une instance de `TextInputType`. Cette classe possède plusieurs constantes :

- `datetime` : clavier optimisé pour la saisie d'heures et de dates

```
keyboardType: TextInputType.datetime,
```

- `emailAddress` : clavier optimisé pour la saisie d'adresse e-mail

```
keyboardType: TextInputType.emailAddress,
```

- `multiline` : clavier optimisé pour la saisie multiligne

```
keyboardType: TextInputType.multiline,
```

- `number` : clavier optimisé pour la saisie de nombre

```
keyboardType: TextInputType.number,
```

- `phone` : clavier optimisé pour la saisie de numéro de téléphone

```
keyboardType: TextInputType.phone,
```

– `text` : clavier optimisé pour la saisie de texte simple

```
keyboardType: TextInputType.text,
```

– `url` : clavier optimisé pour la saisie d'URL

```
keyboardType: TextInputType.url,
```

2.4 TextEditingController

Dans les deux cas précédents (`TextField` et `CupertinoTextField`), la capture de la saisie en temps réel était basée sur la méthode `onChanged`. Cette technique est simple et fonctionne parfaitement.

Une autre façon de procéder, un peu plus élaborée mais aussi plus puissante, consiste à mettre en place un contrôleur qui va fonctionner sur le principe du *pattern* Observateur.

Des abonnements peuvent être réalisés sur celui-ci. En cas de changement, il informera ses abonnés.

Pour reprendre le premier exemple mettant en place un champ `TextField`, un contrôleur est intégré. La structure de base reste la même, à savoir un `TextField` puis un `Container` affichant les modifications.

```
class _MyHomePageState extends State<MyHomePage> {
 TextStyle _textStyleDefault = TextStyle(
   color: Colors.white,
 );

 TextStyle _textStyleSaisie = TextStyle(
   color: Colors.amber,
   fontWeight: FontWeight.bold,
 );


 String _tf = 'Aucun texte saisi';


 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
```

© Editions ENI - All rights reserved

```
body: Padding(
  padding: const EdgeInsets.all(20.0),
  child: Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          'Formulaire',
          style: TextStyle(
              fontWeight: FontWeight.bold,
              fontSize: 40,
              color: Colors.blue),
        ),
        Divider(),
        Padding(padding: EdgeInsets.only(bottom: 40)),
        TextField(
          decoration: InputDecoration(border: OutlineInputBorder()),
        ),
        Padding(padding: EdgeInsets.only(bottom: 20)),
        Container(
          width: MediaQuery.of(context).size.width * 0.9,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceAround,
            children: <Widget>[
              Padding(padding: EdgeInsets.only(top: 10)),
              Text(
                '1- Texte en cours de saisie dans TextField : ',
                style: _textStyleDefault,
              ),
              Text(
                '$_tf',
                style: _textStyleSaisie,
              ),
              Padding(padding: EdgeInsets.only(bottom: 10)),
            ],
          ),
          decoration: BoxDecoration(
              color: Colors.blue,
              borderRadius: BorderRadius.circular(5),
              boxShadow: [
                BoxShadow(
                    color: Colors.black,
                    blurRadius: 25.0,
                    spreadRadius: 5.0,
                    offset: Offset(0, 15))
              ]),
        )
      ],
    ),
```

```
        ),
      ),
    );
  }
}
```

Dans ce code, seul le remplacement de `onChanged` va être montré. Toutefois, il serait aisé de faire de même pour `onSubmitted`.

La première étape est de créer le contrôleur :

```
final _controller = TextEditingController();
```

Il faut maintenant réaliser les abonnements. Cette étape peut être inscrite dans le `initState`. Cette méthode se déclenche une fois, au moment de l'initialisation de l'écran :

```
void initState() {
 super.initState();
 _controller.addListener(_afficherLaValeurDuChamp);
}
```

Dans le code ci-dessus, un abonnement est réalisé. L'abonné (*listener*) est une méthode privée nommée `_afficherLaValeurDuChamp`. Avant de consulter le contenu de cette dernière, il faut prévoir la méthode qui va supprimer le contrôleur ainsi que tous ses effets. Il s'agit de `dispose` :

```
void dispose() {
 _controller.dispose();
 super.dispose();
}
```

Ces étapes de configuration terminées, il est temps de regarder de plus près la méthode `_afficherLaValeurDuChamp` :

```
_afficherLaValeurDuChamp() {
 setState(() {
   _tf = _controller.text;
 });
}
```

Cette méthode est assez simple. Elle met à jour la variable `_tf` par l'intermédiaire de la méthode `text` du `controller`. L'opération se réalise comme à l'accoutumée dans un `setState`.

© Editions ENI - All rights reserved

Il reste une étape cruciale : établir le lien entre le widget `TextField` et ce contrôleur. Dans le constructeur du champ, il existe un paramètre `controller`, c'est parfait ! Il n'y a plus qu'à y affecter celui qui a été créé un peu plus haut :

```
TextField(
 controller: _controller,
 decoration: InputDecoration(border: OutlineInputBorder()),
),
```

Avec cette façon de procéder, il n'y a donc plus besoin de `onChanged`.

Les avantages sont nombreux. Cela permet d'avoir un meilleur contrôle sur l'évolution du champ et de son contenu. La phase d'initialisation permet en outre de définir une valeur de départ également :

```
void initState() {
 super.initState();
 _controller.addListener(_afficherLaValeurDuChamp);
 _controller.value = TextEditingValue(text: '$_tf');
}
```

3. Dates et heures

Dans les formulaires, il est courant d'avoir besoin de récupérer des dates et/ou des heures. De nouveau, ces widgets se déclinent dans les deux styles, Material Design et iOS.

3.1 Material Design

3.1.1 Date

La mise en place d'un `DatePicker` comporte plusieurs étapes. Le widget qui va être utilisé est `showDatePicker`. Il fonctionne donc comme les pop-ups évoqués dans le chapitre Pop-up et navigation. Il va falloir une fonction de type `Future` pour gérer cela.

Mais avant d'examiner le code de cette dernière, il faut préparer le squelette de l'application. Il se base sur une classe `StatefulWidget` puisque la gestion des états va servir à mettre à jour l'écran en fonction de la date récupérée.

Pour déclencher l'ouverture de la fenêtre dédiée au choix de la date, un widget `RaisedButton` est intégré. Juste en dessous, une zone de texte viendra afficher la date qui aura été choisie.

Cette application ressemble à cela :

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData.light(),
     home: MyHomePage(title: 'Flutter Demo Date'),
     );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;


 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
```

© Editions ENI - All rights reserved

```
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          RaisedButton(
            child: Text('Ouvrir un DatePicker'),
            color: Colors.blue,
            textColor: Colors.white,
            elevation: 8,
            onPressed: () {
              _selectionDate();
            }),
          Padding(padding: EdgeInsets.only(bottom: 20)),
          Text(
            'Voici la date choisie:',
          ),
          Padding(padding: EdgeInsets.only(bottom: 20)),
          ],
        ),
      ),
    );
  }
}
```

La propriété `onPressed` du bouton lance une méthode intitulée `_selectionDate` qu'il convient de créer. En dehors du `build`, ce code est ajouté :

```
Future<Null> _selectionDate() async {
 DateTime _dateChoisie = await showDatePicker(
   context: context,
   initialDate: DateTime.now(),
   firstDate: DateTime(1900),
   lastDate: DateTime(2030),
 );

}
```

La fonction de type `Future` est asynchrone, car elle va attendre que l'utilisateur fasse un choix dans le pop-up `DatePicker`. Ce choix va être assigné à une variable de type `DateTime` nommée `_dateChoisie`. Le temps que ce retour arrive, l'exécution du `showDatePicker` se réalise grâce à `await`. Le widget possède un constructeur avec plusieurs paramètres obligatoires. Le paramètre `context` qui correspond au contexte courant. Une date initiale (`initialDate`) sur laquelle le `DatePicker` va se positionner au démarrage. Enfin, deux bornes avec `firstDate` pour la date la plus ancienne acceptée et `lastDate` pour la date la plus récente. Ces trois derniers paramètres acceptent des instances de `DateTime`. Dans l'exemple la date de départ (`initialDate`) est fixée à aujourd'hui grâce à `DateTime.now()`. Quant aux bornes minimum et maximum, elles sont fixées respectivement à 1900 (le 01/01/1900 pour être exact) et 2030 (01/01/2030).

Il manque dans cette application la zone de texte qui doit être mise à jour au moment du choix de la date. Il faut définir au préalable une variable susceptible d'évoluer.

```
var _date = null;
```

Dans la méthode `_selectionDate`, un `setState` est intégré pour mettre celle-ci à jour :

```
Future<Null> _selectionDate() async {
 DateTime _dateChoisi = await showDatePicker(
   context: context,
   initialDate: DateTime.now(),
   firstDate: DateTime(1900),
   lastDate: DateTime(2030),
 );

 if (_dateChoisie != null) {
   setState(() {
     _date = _dateChoisie;
   });
 }
}
```

© Editions ENI - All rights reserved

La variable étant paramétrée, il ne reste plus qu'à l'intégrer dans l'affichage de l'écran. À cette occasion, des conditions ternaires sont utilisées pour la variable elle-même d'une part, ainsi que pour son style d'autre part. Elle va se situer entre le texte *'Voici la date choisie'* et le dernier `Padding`.

```
Text(
 _date == null
      ? 'Aucune date choisie'
      : '${_date.day}/${_date.month}/${_date.year}',
 style: _date == null
      ? TextStyle(color: Colors.blue)
      : TextStyle(
          color: Colors.blue,
          fontWeight: FontWeight.bold,
          fontSize: 40),
),
```

Si la date récupérée est `null`, alors un texte de remplacement est prévu. Dans ce cas, le style est simplifié notamment au niveau de la taille du texte qui reste celle prévue par défaut.

En revanche, si une valeur remonte, la variable étant de type `DateTime`, il est possible de récupérer les valeurs du jour (`day`), du mois (`month`) et de l'année (`year`) séparément pour formater à notre guise la présentation. Parallèlement, la police grandit et se met en gras.

Il reste toutefois un dernier point qui peut se révéler important : le pop-up affiche un calendrier qui n'est pas au format français par défaut. Il est possible de forcer ce comportement. Cela nécessite l'ajout du package **localizations**. Il convient de modifier le fichier *pubspec.yaml* de cette façon :

```
dependencies:
 flutter:
   sdk: flutter

 flutter_localizations:
   sdk: flutter

 # The following adds the Cupertino Icons font to your application.
 # Use with the CupertinoIcons class for iOS style icons.
 cupertino_icons: ^0.1.2
```

Pour récupérer cette dépendance, il ne faut pas oublier de cliquer sur **Packages get**.

Dans le fichier *main.dart*, il est maintenant nécessaire de placer cet import :

```
import 'package:flutter_localizations/flutter_localizations.dart';
```

Puis, dans le `build` de la classe `MyApp` et, pour être vraiment précis, dans le constructeur de `MaterialApp`, il faut appeler deux paramètres nommés `localizationsDelegates` et `supportedLocales` :

```
class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData.light(),
     home: MyHomePage(title: 'Flutter Demo Date'),
     localizationsDelegates: [
       GlobalMaterialLocalizations.delegate,
       GlobalWidgetsLocalizations.delegate,
     ],
     supportedLocales: [
       const Locale('fr', 'FR'),
     ],
   );
 }
}
```

© Editions ENI - All rights reserved

Avec cette dernière configuration, le `DatePicker` va être en français. Après avoir cliqué sur le bouton, le rendu est le suivant :

Figure 5 - DatePicker

Une fois la date choisie et le bouton **OK** cliqué, le pop-up se ferme et la date récupérée apparaît :

Figure 6 - Récupération de la date

3.1.2 Heure

Il existe un outil similaire pour l'heure, `showTimePicker`. Son implémentation se réalise de la même façon que pour la date.

Pour commencer, il faut évidemment un bouton qui va permettre de déclencher l'ouverture du widget :

```
RaisedButton(
   child: Text('Ouvrir un TimePicker'),
   color: Colors.green,
   textColor: Colors.white,
   elevation: 8,
   onPressed: () {
```

© Editions ENI - All rights reserved

```
      _selectionHeure();
    }),
```

Le bouton est intégré directement sous le premier servant pour la date. Cette fois-ci, la méthode appelée sur `onPressed` est `_selectionHeure`:

```
Future<Null> _selectionHeure() async {
 TimeOfDay _heureChoisi =
     await showTimePicker(context: context, initialTime: TimeOfDay.now());

 if (_heureChoisi != null) {
   setState(() {
     _heure = _heureChoisi;
   });
 }
}
```

La façon de procéder reste la même, au travers d'une fonction `Future`. Cependant la variable qui va être récupérée est maintenant de type `TimeOfDay`.

Le constructeur de `showTimePicker` est un peu plus simple et ne comporte que deux paramètres obligatoires : le contexte (`context`) et l'heure initiale (`initialTime`).

Comme auparavant, cette fonction `Future` fournit l'occasion de placer un `setState` mettant à jour une variable qui servira à l'affichage. Cette dernière est nommée `_heure`. Elle est déclarée préalablement de cette façon :

```
TimeOfDay _heure = null;
```

Concernant l'affichage de la valeur récupérée, le procédé est le même que pour la date et ce code est donc ajouté à la suite du dernier `Padding` :

```
Text(
 'Voici l\'heure choisie:',
),
Text(
 _heure == null
     ? 'Aucune heure choisie'
     : '${_heure.format(context)}',
 style: _heure == null
     ? TextStyle(color: Colors.green)
     : TextStyle(
         color: Colors.green,
```

```
            fontWeight: FontWeight.bold,
            fontSize: 40),
  ),
```

La variable `_heure` étant de type `TimeOfDay`, il est possible de lui appliquer la méthode `format` pour lui donner le style voulu.

Le pop-up prend cette forme :

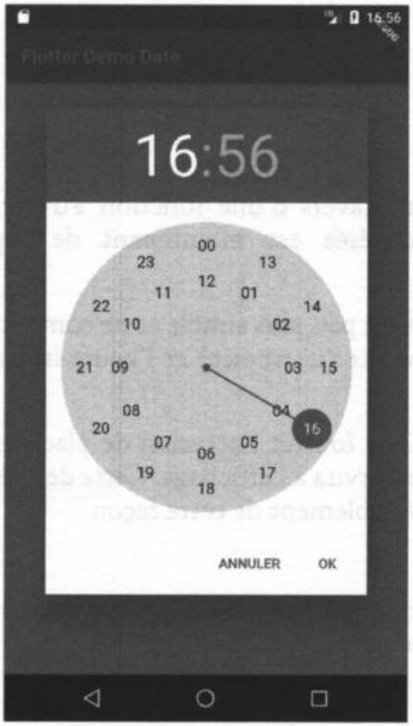


Figure 7 - TimePicker

© Editions ENI - All rights reserved

Une fois l'heure choisie et le bouton **OK** cliqué, le retour sur l'écran principal permet de vérifier que l'heure en question est bien récupérée et traitée :

Figure 8 - Récupération de l'heure

3.2 Cupertino

3.2.1 Date

Le widget `CupertinoDatePicker` s'implémente un peu différemment. L'idéal étant de le faire apparaître dans un widget `BottomSheet`. Le constructeur propose plusieurs paramètres intéressants. D'abord le `mode`. Il permet de choisir entre `dateTime`, `time` et `date`. Dans cette section, c'est ce dernier qui va être étudié. Il est évidemment possible de choisir une date de départ (`initialDateTime`), mais aussi de borner le choix avec une date minimum (`minimumDate`) et une date maximum (`maximumDate`).

Un avantage non négligeable est qu'il offre également une méthode nommée `onDateTimeChanged` qui est le lieu idéal pour l'insertion d'un `setState` puisque tous les changements seront captés ici.

Pour découvrir le widget, une nouvelle application est créée. Elle est très semblable à la dernière. Elle se compose donc d'un bouton qui va déclencher l'ouverture du `CupertinoDatePicker` puis d'un texte qui affichera la valeur récupérée. Voici le squelette de cette application :

```
import 'package:flutter/material.dart';
import 'package:flutter/cupertino.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo CupertinoDatePicker'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text(widget.title),
     ),
     body: Center(
       child: Column(
```

© Editions ENI - All rights reserved

```
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            RaisedButton(
                child: Text('Ouvrir un DatePicker'),
                color: Colors.blue,
                textColor: Colors.white,
                elevation: 8,
                onPressed: () {
                  showModalBottomSheet(
                      context: context,
                      builder: (BuildContext context) {
                        return Container(
                          height: MediaQuery.of(context).size.height/3,
                          child: _datePicker(),
                        );
                      });
                }
            ),
            Padding(padding: EdgeInsets.only(bottom: 20)),
            Text(
                'Voici la date choisie:',
            ),
            Padding(padding: EdgeInsets.only(bottom: 20)),
            ],
        ),
      ),
      backgroundColor: Colors.white,
    );
  }
}
```

Le widget RaisedButton appelle avec onPressed le constructeur du widget showModalBottomSheet. Le builder retourne un objet Container qui hébergera dans child un widget nommé _datePicker.

Voici ce widget :

```
Widget _datePicker() {
 return CupertinoDatePicker(
     mode: CupertinoDatePickerMode.date,
     initialDateTime: DateTime.now(),
     minimumDate: DateTime(1900),
     maximumDate: DateTime(2030),
     onDateTimeChanged: (value) {
       setState(() {
         _date = value;
       });
     });
}
```

Le widget retourné est de type `CupertinoDatePicker`. Sa configuration reste la même que dans l'exemple précédent, tant pour la date d'initialisation que pour les bornes minimums et maximums.

L'appel à `setState` est inclus dans `onDateTimeChanged`. Il met à jour la variable `_date` déclarée préalablement comme suit :

```
DateTime _date = null;
```

Il reste à afficher la date récupérée. Le code de la présentation est inséré au-dessus du dernier `Padding` :

```
Text(
 _date == null
     ? 'Aucune date choisie'
     : '${_date.day}/${_date.month}/${_date.year}',
 style: _date == null
     ? TextStyle(color: Colors.blue)
     : TextStyle(
         color: Colors.blue,
         fontWeight: FontWeight.bold,
         fontSize: 40),
),
```

© Editions ENI - All rights reserved

Pour l'affichage, il n'y a pas de différence avec ce qui a été décrit précédemment. Il n'y a donc plus qu'à découvrir le rendu :

Figure 8 - CupertinoDatePicker

3.2.2 Heure

La mise en place du TimePicker est très semblable. Le paramètre `mode` va être modifié pour prendre la valeur `time`. Le paramètre `minuteInterval` est passé à 1 afin de pouvoir choisir une heure à la minute près. Tandis que `use24hFormat` est valorisé à `true`, dans le but d'obtenir un format d'heure sur 24 heures.

Le reste diffère peu pour l'instant. Pour commencer, il faut intégrer un bouton qui permette d'ouvrir le widget :

```
RaisedButton(
   child: Text('Ouvrir un TimePicker'),
   color: Colors.green,
   textColor: Colors.white,
   elevation: 8,
   onPressed: () {
     showModalBottomSheet(
         context: context,
         builder: (BuildContext context) {
           return Container(
             height: MediaQuery.of(context).size.height/3,
             child: _timePicker(),
           );
         });
   }
),
```

De la même façon que pour le DatePicker, un appel à `showModalBottomSheet` ouvre un `Container` dont l'enfant (`child`) est un widget. Celui utilisé pour l'heure s'appelle `_timePicker`. Voici son code :

```
Widget _timePicker() {
 return CupertinoDatePicker(
     minuteInterval: 1,
     mode: CupertinoDatePickerMode.time,
     use24hFormat: true,
   initialDateTime: DateTime.now(),
   onDateTimeChanged: (value) {
       setState(() {
         _heure = TimeOfDay.fromDateTime(value);
       });
   },
   );
}
```

Hormis les paramètres décrits plus haut, l'`initialDateTime` est toujours laissé sur `DateTime.now`. Ce qui a pour effet de positionner le TimePicker sur l'heure courante (attention, si l'application est lancée dans l'émulateur, l'heure chargée est bien celle de ce dernier).

© Editions ENI - All rights reserved

Il est possible de remarquer une différence dans la définition du `setState` de `onDateTimeChanged`. En effet, la valeur qui évolue dans `onDateTimeChanged` est de type `DateTime`. Or, la variable qui va servir pour l'affichage est, elle, de type `TimeOfDay`. Heureusement, il existe un constructeur nommé de cette classe qui accepte un paramètre `DateTime` : `TimeOfDay.fromDateTime`.

La déclaration de la variable avait été écrite comme ceci :

```
TimeOfDay _heure = null;
```

Enfin, l'affichage doit prendre en considération cette nouvelle information. Comme dans l'application précédente, un widget `Text` va se charger de cette tâche :

```
Text(
 'Voici l\'heure choisie:',
),
Text(
 _heure == null
     ? 'Aucune heure choisie'
     : '${_heure.hour}:${_heure.minute}',
 style: _heure == null
     ? TextStyle(color: Colors.green)
     : TextStyle(
     color: Colors.green,
     fontWeight: FontWeight.bold,
     fontSize: 40),
),
```

Pour rendre cet exemple plus parlant, voici le rendu qui est obtenu :

Figure 9 - CupertinoDatePicker en mode time

© Editions ENI - All rights reserved

4. Switch

Le *switch* est communément considéré comme un interrupteur. Il possède deux positions correspondant à un état faux et un autre vrai. Il est parfait pour permettre à l'utilisateur de faire des choix simples.

4.1 Material Design

Le widget `Switch` est celui qui correspond au style Material Design. Son constructeur prend deux paramètres obligatoires. Il faut, dans un premier temps définir une valeur (`value`), en l'occurrence un booléen. Puis, la propriété `onChanged` doit être complétée. Il sera tout à fait possible d'y insérer un appel à `setState`.

D'autres paramètres optionnels sont également présents. Parmi ceux les plus utilisés, il y a `activeColor` qui permet de définir une couleur précise quand le `Switch` est actif. La couleur choisie est opérante pour le curseur tout comme pour la piste (*track*) qui se décline dans une note plus claire. Si cette déclinaison ne convient pas, il est toujours possible d'en spécifier une autre avec `activeTrackColor`. À l'inverse, quand l'utilisateur bascule le `Switch` dans une position inactive, la couleur du curseur pourra être modifiée grâce à `inactiveThumbColor` et la *track* avec `inactiveTrackColor`.

Le curseur peut même être remplacé par une image via `activeThumbImage` et `inactiveThumbImage`.

Pour mettre en lumière un exemple, une application classique est mise en place. Elle possède une classe qui étend `StatefulWidget` afin que l'état puisse être géré par la suite.

Un booléen, `choixSwitch`, est déclaré au début de la classe `_MyHomePageState`. Il va servir à vérifier la valeur du `Switch` et à la confirmer à l'écran.

```
bool _choixSwitch = false;
```

Dans le corps de la classe, au sein de la méthode `build`, le `Switch` est inséré :

```
@override
 Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Choix de l\'utilisateur',
            style: TextStyle(
              color: Colors.blueGrey,
              fontSize: 38
            ),
          ),
          Divider(),
          Text(
            'Switch',
            style: TextStyle(fontWeight: FontWeight.bold, fontSize: 30),
          ),
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Switch(
                value: _choixSwitch,
                activeColor: Colors.green,
                activeTrackColor: Colors.amber,
                inactiveTrackColor: Colors.red[200],
                inactiveThumbColor: Colors.red,
                onChanged: (value) {
                  setState(() {
                    _choixSwitch = value;
                  });
                }),
            ],
          ),
          Text(_choixSwitch ? 'Pour' : 'Contre')
        ],
      ),
    ),
  );
 }
```

© Editions ENI - All rights reserved

Plusieurs paramètres du constructeur sont alimentés pour que l'exemple soit plus parlant. Ainsi, en cas d'inactivité, le widget deviendra rouge (`inactiveThumbColor`) tout comme la *track* avec une tonalité plus claire (`inactiveTrackColor`).

En cas d'activation, la *track* se colore en orange (`activeTrackColor`) et le curseur en vert (`activeColor`).

En dehors de ces aspects graphiques, `value` prend le booléen créé un peu plus haut. L'évolution de ce dernier se réalise dans un appel à `setState` placé dans `onChanged`.

Pour valider l'expérience, un texte est intégré juste en dessous du widget. Une condition ternaire permet de faire varier sa valeur de *'pour'* à *'contre'* en fonction de la valeur du booléen `_choixSwitch`.

Le résultat est plutôt probant :

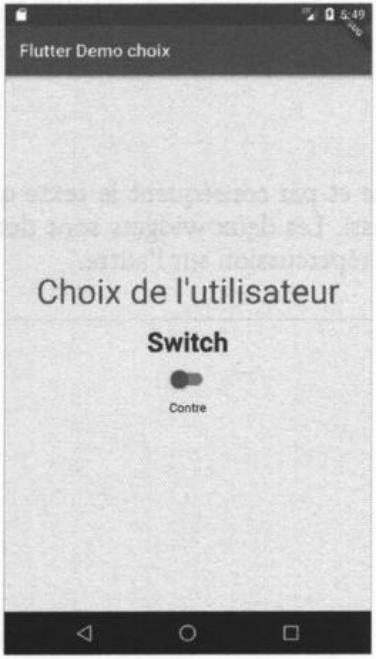


Figure 10 - Switch

4.2 Cupertino

Le *switch* dans le style Cupertino se nomme tout simplement `CupertinoSwitch`. Ce widget est un peu plus simple et ne possède pas autant d'éléments de configuration que celui vu dans le point précédent. De base, le constructeur propose les deux mêmes paramètres obligatoires, `value` et `onChanged`.

En dehors, sur la partie visuelle, le seul élément paramétrable est la couleur en cas d'activation. Cette personnalisation se réalise via `activeColor`.

Pour comprendre son fonctionnement, l'application est enrichie. Le `Switch` Material Design avait été placé volontairement dans une `Row`. Le `CupertinoSwitch` vient donc en second enfant.

```
CupertinoSwitch(
    value: _choixSwitch,
    activeColor: Colors.cyan,
    onChanged: (value) {
      setState(() {
        _choixSwitch = value;
      });
    })
```

Le booléen qui sert de référence est le même et par conséquent le texte qui affiche la valeur du `CupertinoSwitch` aussi. Les deux widgets sont donc liés et le changement d'état de l'un aura une répercussion sur l'autre.

© Editions ENI - All rights reserved

Le rendu est celui-ci :

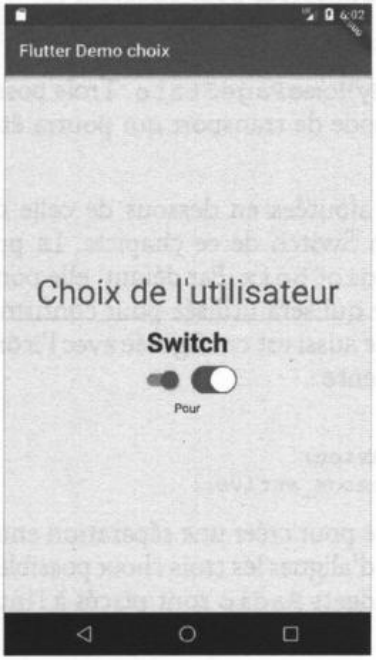


Figure 11 - CupertinoSwitch

5. Radio

Les boutons « radios » sont une autre manière de récupérer un choix émis par l'utilisateur. Ils offrent potentiellement plus de choix qu'un Switch, mais reste limités puisque le choix multiple n'est pas autorisé.

Concernant Flutter, il n'existe qu'un seul widget au style Material, nommé `Radio`. Le constructeur impose trois paramètres obligatoires : `value`, `groupValue` et `onChanged`. La logique est donc proche de celle du `Switch`. La différence principale réside dans la présence du paramètre `groupValue` qui sert à « lier » les occurrences de `Radio` entre elles en les faisant appartenir à un même groupe.

Pour présenter le widget au sein de la même application, une énumération est ajoutée. Elle va servir de base au choix que l'utilisateur peut effectuer :

```
enum RadioChoix { Voiture, Avion, Bateau }
```

Elle doit être placée en dehors de la classe `_MyHomePageState`. Trois possibilités y sont renseignées sur un éventuel mode de transport qui pourra être voiture, avion ou bateau.

Au début de la classe, deux variables sont ajoutées en dessous de celle du booléen `_choixSwitch` vu dans la section Switch de ce chapitre. La première prend le type de l'énumération soit `RadioChoix`. Par défaut, elle porte sur la valeur avion. La seconde est une icône qui sera utilisée pour confirmer visuellement le choix fait par l'utilisateur. Elle aussi est configurée avec l'icône de l'avion pour être cohérente avec la précédente :

```
bool _choixSwitch = false;
RadioChoix _choixRadio = RadioChoix.Avion;
Icon _iconRadio = Icon(Icons.airplanemode_active);
```

Dans l'appel à `build`, un `Divider` est inséré pour créer une séparation entre le `Switch` et la zone des radio boutons. Afin d'aligner les trois choix possibles, un widget `Row` est mis en place. Les trois widgets `Radio` sont placés à l'intérieur.

Chacun des widgets prend une valeur (`value`) qui correspond à l'une de l'énumération. Le regroupement (`groupValue`) se fait au travers de la variable `RadioChoix _choixRadio`. Le paramètre `onChanged` implémente `setState` qui mettra à jour cette variable ainsi que l'icône :

```
Divider(),
Text(
 'Radio',
 style: TextStyle(fontWeight: FontWeight.bold, fontSize: 30),
),
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
   Radio(
       value: RadioChoix.Voiture,
       groupValue: _choixRadio,
       activeColor: Colors.green,
       onChanged: (value) {
         setState(() {
```

© Editions ENI - All rights reserved

```
            _choixRadio = value;
            _iconRadio= Icon(Icons.directions_car);
          });
        }),
    Text('Voiture', style: _choixRadio == RadioChoix.Voiture ?
TextStyle(color: Colors.green, fontWeight: FontWeight.bold) : TextStyle()),
    Radio(
        value: RadioChoix.Avion,
        groupValue: _choixRadio,
        activeColor: Colors.green,
        onChanged: (value) {
          setState(() {
            _choixRadio = value;
            _iconRadio = Icon(Icons.airplanemode_active);
          });
        }),
    Text('Avion', style: _choixRadio == RadioChoix.Avion ?
TextStyle(color: Colors.green, fontWeight: FontWeight.bold) : TextStyle()),
    Radio(
        value: RadioChoix.Bateau,
        groupValue: _choixRadio,
        activeColor: Colors.green,
        onChanged: (value) {
          setState(() {
            _choixRadio = value;
            _iconRadio = Icon(Icons.directions_boat);
          });
        }),
    Text('Bateau', style: _choixRadio == RadioChoix.Bateau ?
TextStyle(color: Colors.green, fontWeight: FontWeight.bold) : TextStyle()),
  ],
),
```

Pour agrémenter l'aspect visuel, le paramètre `activeColor` est passé sur vert. Pour rappel, en temps normal, il est bleu.

Pour que ce changement de couleur soit encore plus pertinent, un effet du même ordre est appliqué sur les textes qui accompagnent chaque radio bouton. Ainsi, par le biais d'une condition ternaire un objet `TextStyle` modifie le texte en vert et en gras si le radio bouton associé est choisi. Il reste dans un style classique si ce n'est pas le cas.

Enfin, il ne reste plus qu'à intégrer l'icône. Le code suivant est donc placé sous la définition de `Row` des `Radio` :

```
), //Row
_iconRadio,
```

Voici le rendu de cette nouvelle fonctionnalité :

Figure 12 - Radio Bouton

6. Checkbox

Pour compléter les widgets offrant une liberté de choix à un utilisateur, il faut évoquer les `Checkbox`. Contrairement aux boutons radios, les cases à cocher permettent un choix multiple. Il n'y a donc pas de notion de groupe, chacune étant indépendante. De fait, leur constructeur ne possède que deux paramètres obligatoires que sont `value` et `onChanged`.

Bien sûr, il existe d'autres paramètres qui permettent d'améliorer l'esthétique du widget comme `activeColor`.

© Editions ENI - All rights reserved

Pour en faire une démonstration, toujours sur la même application, une variable va être ajoutée. Il s'agit d'une Map qui, pour rappel, a la particularité de fonctionner avec un système de clé/valeur. Cela va s'avérer utile puisque le paramètre value attend un booléen. Concernant la clé, il va être confortable de réutiliser l'énumération précédemment définie. L'objet Map compose donc comme ceci :

```
Map _mapTransport = {
 RadioChoix.Voiture: false,
 RadioChoix.Avion: false,
 RadioChoix.Bateau: false,
};
```

Pour l'instant, toutes les valeurs sont passées à false. Aucune Checkbox ne sera cochée. Celles-ci sont placées après un Divider au sein d'une Row :

```
Divider(),
Text('Checkbox',
 style: TextStyle(fontWeight: FontWeight.bold, fontSize: 30),
),
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
   Checkbox(
       activeColor: Colors.green,
       value: _mapTransport[RadioChoix.Voiture],
       onChanged: (value) {
         setState(() {
           _mapTransport[RadioChoix.Voiture] = value;
         });
       }
   ),
   Text('Voiture', style: _mapTransport[RadioChoix.Voiture] ?
TextStyle(color: Colors.green) : TextStyle()),
   Checkbox(
       activeColor: Colors.green,
       value: _mapTransport[RadioChoix.Avion],
       onChanged: (value) {
         setState(() {
           _mapTransport[RadioChoix.Avion] = value;
         });
       }
   ),
```

```
    Text('Avion', style: _mapTransport[RadioChoix.Avion] ?
TextStyle(color: Colors.green) : TextStyle()),
    Checkbox(
        activeColor: Colors.green,
        value: _mapTransport[RadioChoix.Bateau],
        onChanged: (value) {
          setState(() {
            _mapTransport[RadioChoix.Bateau] = value;
          });
        }
    ),
    Text('Bateau', style: _mapTransport[RadioChoix.Bateau] ?
TextStyle(color: Colors.green) : TextStyle()),
  ],
),
```

Chaque widget `Checkbox` est accompagné d'un texte. Si la case est cochée, sa couleur passe au vert grâce au paramètre `activeColor`. Chaque `value` se voit affecter une valeur de l'objet `Map`. Dans `onChanged`, cette valeur évolue. Les textes qui accompagnent les `Checkbox` changent de style en fonction de l'évolution des booléens de l'objet `Map`. Pour compléter l'aspect graphique, des icônes sont ajoutées à la suite. De même que pour les `Checkbox` et les textes, elles seront colorées si la case est cochée et grisées si ce n'est pas le cas. Pour éviter de prendre trop de place à l'écran, elles sont placées dans une nouvelle occurrence de `Row` :

```
Row(
 mainAxisAlignment: MainAxisAlignment.center,
 children: <Widget>[
   Icon(Icons.directions_car, color: _mapTransport[RadioChoix.Voiture] ?
Colors.green : Colors.grey[300],),
   Icon(Icons.airplanemode_active, color: _mapTransport[RadioChoix.Avion] ?
Colors.green : Colors.grey[300],),
   Icon(Icons.directions_boat, color: _mapTransport[RadioChoix.Bateau] ?
Colors.green : Colors.grey[300],),
 ],
),
```

© Editions ENI - All rights reserved

Voici l'écran avec l'ajout de ces derniers composants avec un choix qui se porte sur **Voiture** et **Avion** :

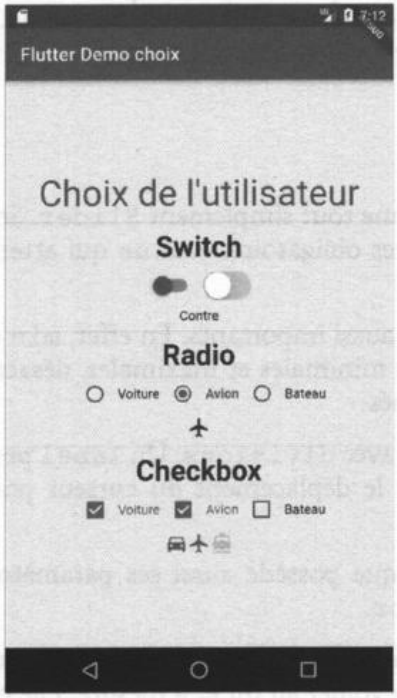


Figure 13 - Checkbox

7. Slider

Les *sliders* sont un autre type de composant de formulaire. Ils permettent à l'utilisateur de choisir une valeur sur une échelle définie. Cette fois-ci, deux widgets remplissent ce rôle, chacun avec le style lié à sa plateforme.

7.1 Material Design

Le widget au style Material Design se nomme tout simplement `Slider`. Son constructeur n'impose que deux paramètres obligatoires, `value` qui attend une valeur de type double, et `onChanged`.

Cependant, d'autres paramètres sont tout aussi importants. En effet, `min` et `max`, qui permettent de définir des valeurs minimales et maximales, désactiveront le `Slider` s'ils ne sont pas renseignés.

Au-delà, il est possible de définir des paliers avec `divisions`. Un `label` peut aussi être inséré. Il viendra accompagner le déplacement du curseur pour donner une indication.

Enfin, comme d'habitude, l'aspect graphique possède aussi ses paramètres comme `activeColor` et `inactiveColor`.

Pour tester le widget dans la même application, il faut au préalable créer une variable de type `double`. C'est elle qui va évoluer au fur et à mesure. Elle est insérée au début de `_MyHomePageState`.

```
double _choixSlider = 5.0;
```

Dans l'appel à `build`, un `Divider` permet de séparer le code précédent du `Slider`. Dans ce dernier, la variable `choixSlider` est affectée à la propriété `value`. Le `label` est affecté également, mais cette fois sous la forme d'une chaîne de caractères. En ce qui concerne les valeurs minimum et maximum, elles sont définies respectivement sur 0.0 et 10.0. Le `Slider` est découpé en 10 via `divisions`.

Par défaut, le `Slider` apparaît en bleu soutenu pour la partie active et en bleu plus clair pour celle inactive. Grâce à `activeColor` et `inactiveColor`, ces couleurs sont changées pour du vert.

© Editions ENI - All rights reserved

Enfin, onChanged mettra à jour la valeur de _choixSlider par l'intermédiaire d'un appel à setState.

```
Divider(),
Text('Slider',
 style: TextStyle(fontWeight: FontWeight.bold, fontSize: 30),
),
Slider(
   value: _choixSlider,
   label: '$_choixSlider',
   activeColor: Colors.green,
   inactiveColor: Colors.green[100],
   min: 0.0,
   max: 10.0,
   divisions: 10,
   onChanged: (value) {
     setState(() {
       _choixSlider = value;
     });
   },
),
```

Comme d'habitude, afin de vérifier la récupération du choix de l'utilisateur, une zone de texte est placée en dessous. Elle va afficher la valeur courante du Slider et sa couleur va passer du rouge au vert en fonction d'un seuil défini à 5.0 grâce à une structure ternaire :

```
Text(
 'Valeur du Slider : $_choixSlider',
 style: TextStyle(
   color: _choixSlider <5 ? Colors.red : Colors.green
 ),
),
```

Le résultat obtenu est le suivant :

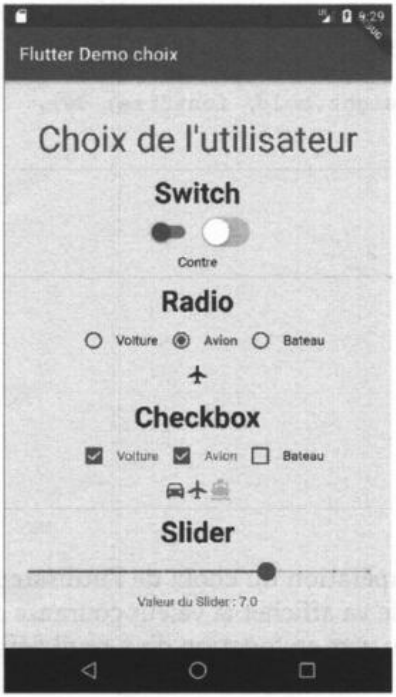


Figure 14 - Slider

7.2 Cupertino

Le *slider* qui correspond au style Cupertino est le widget `CupertinoSlider`. Son implémentation est identique à celle du `Slider`.

Dans l'application, il est inséré entre le `Slider` et le dernier `Text`. La variable `_choixSlider` va être réutilisée. Les deux *sliders* évolueront donc concomitamment.

© Editions ENI - All rights reserved

Voici le code de ce widget :

```
Container(
 width: MediaQuery.of(context).size.width*0.92,
 child: CupertinoSlider(
     value: _choixSlider,
     activeColor: Colors.green,
     thumbColor: Colors.green,
     min: 0.0,
     max: 10.0,
     divisions: 10,
     onChanged: (value) {
       setState(() {
         _choixSlider = value;
       });
     },
 ),
),
```

Le `CupertinoSlider` est placé dans un `Container`. En temps normal, sa largeur est plus réduite que celle d'un `Slider` Material Design. Pour faire en sorte de les rendre similaires, la mise en place d'un `Container` permet de jouer sur sa dimension.

Les paramètres du widget sont identiques à l'exemple précédent. Le minimum (`min`) est à 0 tandis que le maximum (`max`) est à 10. Les `divisions` sont aussi identiques.

Les couleurs par défaut sont remplacées par la couleur orange, ce qui permet de le distinguer du `Slider` situé au-dessus.

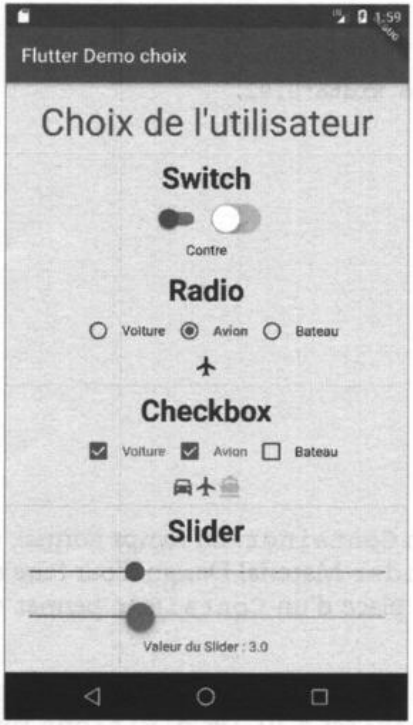
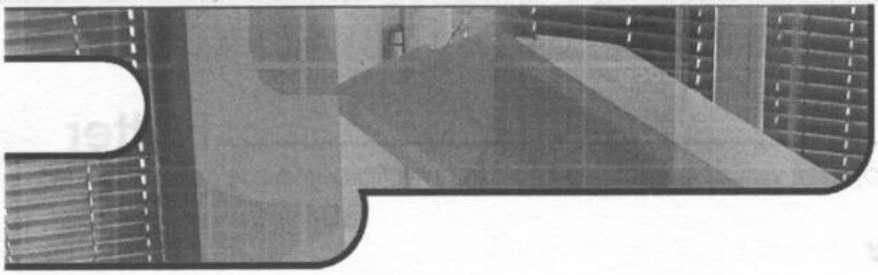


Figure 15 - CupertinoSlider

© Editions ENI - All rights reserved



Chapitre 10
Listes et grilles

1. Introduction

Il existe un élément très important qui n'a pas encore été abordé dans l'ouvrage. Comment faire apparaître des collections ?

L'exposition d'un grand nombre d'éléments à l'écran n'est jamais évidente et peut être la source de beaucoup de problèmes. Afin de couvrir ces besoins, l'équipe Flutter met à disposition deux widgets qui s'avèrent très complémentaires.

Le premier est la `ListView` qui, comme son nom l'indique, permet l'affichage d'une collection sous forme de liste. Le chapitre abordera plusieurs façons d'implémenter ce widget ainsi que différentes méthodes de présentation des items.

Le second widget est la `GridView`. Son nom est également explicite. Cette fois-ci, il n'est plus question d'un affichage sous forme de liste, mais d'une grille. Celle-ci sera composée de vignettes qui peuvent, elles aussi, revêtir différentes formes.

2. ListView

La `ListView` est un outil très pratique pour afficher une grande quantité d'informations à l'écran. Dans le chapitre précédent, les erreurs liées aux dépassements avaient été évoquées rapidement. Pour revenir sur ce point, si la quantité d'informations à afficher dépasse la taille de l'écran, une erreur se produit. Pour remédier à cela, un widget `SingleChildScrollView` avait été utilisé. Toutefois, cela ne constitue pas une solution durable.

La solution préconisée donc est l'utilisation du `ListView`. Tel quel, le constructeur permet d'insérer une liste de widgets enfants. Cette solution est idéale pour des listes courtes. En effet, l'ensemble des enfants, visibles ou non, sera construit dès le départ. Cette charge peut s'avérer très lourde si la liste est longue.

2.1 ListView.builder

Pour des quantités d'informations plus conséquentes, il vaut mieux utiliser le constructeur nommé `ListView.builder`. Il prend un paramètre de type `IndexedWidgetBuilder` qui va créer les enfants à la demande, au fur et à mesure qu'ils seront visibles, donc pendant le défilement de la liste. C'est sur ce widget que le prochain exemple va porter.

Mais avant cela, il convient de détailler un peu les éléments qui vont permettre de construire cette `ListView`.

Dans le constructeur nommé, il n'y a qu'un paramètre obligatoire nommé `itemBuilder`. Comme évoqué plus haut, celui-ci attend une instance de la classe `IndexedWidgetBuilder`. Ce widget réclame deux éléments : un `BuildContext` et un `index`.

Le second paramètre important à prendre en considération est `itemCount`. Il permet de définir la taille de la `ListView`.

© Editions ENI - All rights reserved

Avant de mettre en place cela, il faut se doter d'une collection. Dans ce but, une classe `Voiture` est créée. Elle prend une forme simple et possède seulement trois attributs `marque`, `modele` et `image`, ainsi qu'un constructeur :

```
class Voiture {
 String marque;
 String modele;
 Image image;

 Voiture(this.marque, this.modele, this.image);
}
```

L'application est classique. La classe `MyHomePage` va étendre `StatefulWidget`. Théoriquement, une `ListView` peut être implémentée sans gestion des états. Toutefois, un autre widget inconnu pour l'heure va servir également. Il s'agit d'une surprise qui sera évoquée juste après. En attendant, une liste de voitures est déclarée au début de la classe `_MyHomePageState`. C'est elle qui va fournir les informations à afficher.

```
List<Voiture> _voitures = [
 Voiture('Renault', 'Twingo', Image.asset('images/car1.png')),
 Voiture('Citroën', 'C5', Image.asset('images/car2.png')),
 Voiture('Ford', 'Focus', Image.asset('images/car3.png')),
 Voiture('Porsche', 'Carrera', Image.asset('images/car4.png')),
 Voiture('Ferrari', 'F340', Image.asset('images/car5.png')),
 Voiture('Renault', 'Twingo', Image.asset('images/car1.png')),
 Voiture('Citroën', 'C5', Image.asset('images/car2.png')),
 Voiture('Ford', 'Focus', Image.asset('images/car3.png')),
 Voiture('Porsche', 'Carrera', Image.asset('images/car4.png')),
 Voiture('Ferrari', 'F340', Image.asset('images/car5.png')),
 Voiture('Renault', 'Twingo', Image.asset('images/car1.png')),
 Voiture('Citroën', 'C5', Image.asset('images/car2.png')),
 Voiture('Ford', 'Focus', Image.asset('images/car3.png')),
 Voiture('Porsche', 'Carrera', Image.asset('images/car4.png')),
 Voiture('Ferrari', 'F340', Image.asset('images/car5.png')),
];
```

La liste est volontairement chargée avec des éléments redondants afin de la rendre plus conséquente. Il est bon de noter que des images sont insérées via `Image.asset`. À cet effet, un dossier *images* a été créé à la racine du projet et cinq images de voitures y ont été placées. Pour qu'elles soient exploitables dans le code ci-dessus, le fichier *pubspec.yaml* a été modifié comme ceci :

```
assets:
- images/
```

Remarque

Pour rappel, cette syntaxe permet de référencer l'ensemble des images qui se trouvent dans le dossier images.

Il est temps de mettre en place la `ListView` par l'appel du constructeur nommé `ListView.builder`.

Voici la méthode `build` de l'application. Elle possède comme d'habitude une `appBar`. Sur le `body`, cette fois, pas besoin de `Center` et de `Column`, la `ListView` est directement mise en place.

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.cyan,
   ),
   body: ListView.builder(
       itemCount: _voitures.length,
       itemBuilder: (context, index) {
         final item = _voitures[index];
         return ...
       }),
 );
}
```

Comme cela avait été dit avant, deux paramètres du constructeur sont utilisés. `itemCount` va définir le nombre d'éléments qui composeront la `ListView`. Puisque c'est la liste `_voitures` qui va être utilisée, sa longueur servira de référence. Elle est obtenue grâce à la propriété `length`.

© Editions ENI - All rights reserved

Quant à l'`itemBuilder`, il reçoit comme `BuildContext` le contexte courant (`context`) ainsi qu'`index`. Ce dernier est un entier. Il va automatiquement débuter à 0 et atteindra une valeur maximale déterminée par `itemCount`.

Le code de l'`itemBuilder` va prévoir un retour qui sera détaillé après.

Avant cela, il convient de remarquer la mise en place d'une variable locale nommée `item`. En effet, la `ListView` va générer automatiquement les différents éléments de la liste comme le ferait un `foreach`. Pour exploiter, à chaque itération, les valeurs se trouvant dans la liste, `item` correspondra à une valeur de `_voitures` correspondant à `index`.

2.2 Dismissible

À cet instant, il serait possible d'alimenter le retour avec un widget servant à afficher les différents éléments des voitures.

Mais avant cela, il est de coutume, en développement mobile, de pouvoir effacer une donnée d'une liste par un geste qui porte le nom de *swipe*. Il s'agit d'un glissement latéral vers la gauche ou la droite. Par défaut, cette fonctionnalité n'est pas intégrée dans une `ListView`.

Pour la mettre en place, il faut s'appuyer sur un autre widget, `Dismissible`.

À cet effet plusieurs paramètres du constructeur sont alimentés. Il faudra tout d'abord une information `Key`. Elle permettra d'identifier chaque élément. Dans l'exemple, les modèles de voiture serviront à cela. La clé va servir à contrôler comment un widget de la liste est remplacé par un autre dans l'arbre des widgets.

Autre élément obligatoire, le paramètre `child`. Cette partie sera vue un peu après.

Pour compléter tout ça, le paramètre `background` se voit assigné un `Container`. Cet aspect est intéressant. Le fait d'utiliser ce widget permet de remplir le fond. Quand l'utilisateur fera glisser un item, le fond sera découvert. Le `Container` va être parfait pour cet usage puisqu'il va combler l'ensemble de cet espace caché. Il est même possible de lui donner une couleur pour améliorer l'expérience utilisateur ou bien encore une icône ou un texte via sa propriété `child`.

Enfin, le widget `Dismissible` possède une propriété intitulée `onDismissed` qui permet de décrire le comportement du widget quand le *swipe* est réalisé. L'idée, ici, est de supprimer un item de la liste. C'est pour cela que la classe `MyHomePage` étend `StatefulWidget`. Un appel à `setState` pourra mettre à jour la liste de voitures. Pour finir, il pourra servir pour afficher une confirmation de la suppression par l'intermédiaire d'une `snackBar`.

```
return Dismissible(
  key: Key(item.modele),
  background: Container(
    child: Icon(
      Icons.delete,
      size: 40,
      color: Colors.white,
    ),
    color: Colors.cyan,
  ),
  onDismissed: (direction) {
    setState(() {
      _voitures.removeAt(index);
    });
    Scaffold.of(context).showSnackBar(SnackBar(
        content: Text('Voiture ${item.modele} supprimée !')));
  },
  child: ...
```

© Editions ENI - All rights reserved

2.3 ListTile

Il ne reste plus qu'à remplir la propriété `child` du widget `Dismissible`. À ce moment, plusieurs solutions peuvent être envisagées. Celles-ci vont être exposées les unes après les autres.

Dans un premier temps, le choix va se porter sur le widget `ListTile` qui est spécialement taillé pour cette tâche. Il est construit pour cela et à ce titre, il possède toutes les options utiles.

Par exemple, il contient un objet `leading`, c'est-à-dire un widget qui vient se placer avant le titre, à sa gauche. Il s'agit le plus souvent d'une icône. Dans le même ordre d'idées, le `trailing`, qui est aussi un widget, mais qui vient se placer après le titre, à droite.

Il est possible de renseigner un titre (`title`) et un sous-titre (`subtitle`). Pour ce dernier, si le texte est un peu trop long, il est conseillé de faire configurer la `ListTile` sur trois lignes en passant le booléen `isThreeLine` à true. Sinon, par défaut, il n'y en aura que deux.

Afin de pouvoir interagir avec chaque `ListTile`, un événement intitulé `onTap` (ainsi qu'un autre nommé `onLongPress`) est prévu.

Dans le cas de la liste de voitures, le modèle servira de titre et la marque de sous-titre. À l'appui sur un item, un pop-up va s'ouvrir pour afficher l'image.

Pour rappel, une variable `item` a été définie au début de `itemBuilder`. C'est sur elle que le reste du code va s'appuyer :

```
final item = _voitures[index];
```

Dans l'élément child du widget Dismissible, la ListTile prend cette forme :

```
child: ListTile(
  title: Text(
    '${item.modele}',
    textAlign: TextAlign.center,
    style: TextStyle(
        fontWeight: FontWeight.bold,
        color: Colors.cyan,
        fontSize: 20),
  ),
  subtitle: Text(
    '${item.marque}',
    textAlign: TextAlign.center,
  ),
  onTap: () {
    showDialog(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            backgroundColor: Colors.cyan,
            title: Text(
              'Photo de la voiture ${item.modele}',
              style: TextStyle(color: Colors.white),
            ),
            content: item.image,
          );
        });
  },
),
```

Dans onTap, la méthode showDialog est appelée. Elle retournera une instance d'AlertDialog. Cette dernière expose un titre (title) qui correspond au modèle de la voiture et en guise de contenu (content) l'image qui lui a été affectée.

© Editions ENI - All rights reserved

Le rendu de la liste apparaît comme ceci :

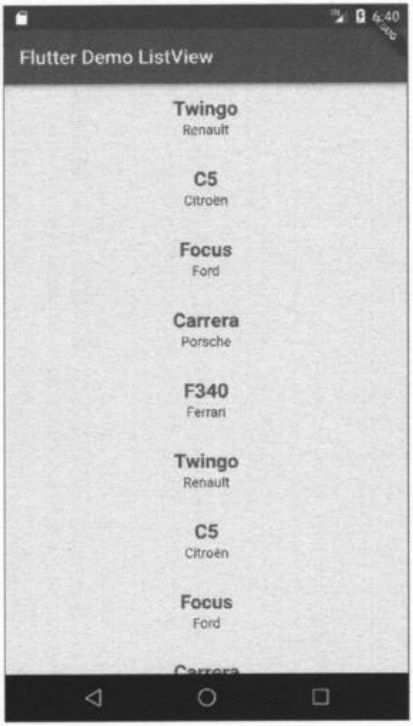


Figure 1 - ListView

Une fois qu'un appui sur un item est détecté, l'`AlertDialog` s'ouvre avec cette apparence :

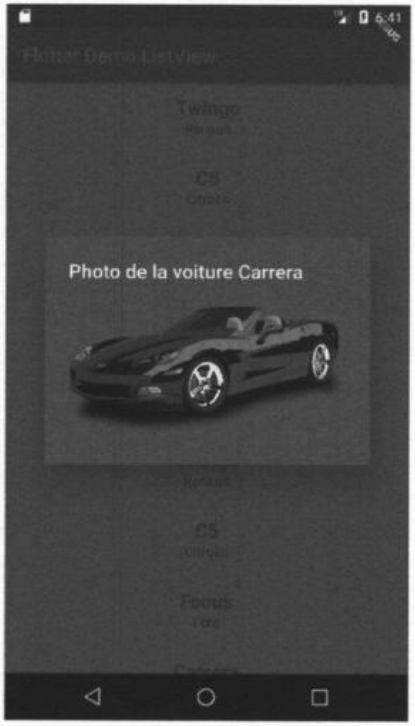


Figure 2 - Exécution de onTap sur un ListTile

© Editions ENI - All rights reserved

2.4 ListView.separated

Sur la figure 1, certains seront peut-être surpris par l'absence de séparation entre les items. Elle peut restreindre ou diminuer la lisibilité et, par conséquent, l'expérience utilisateur.

Les équipes de Flutter ont prévu un second constructeur nommé qui vient répondre à cette problématique. Il s'agit du `ListView.separated`. Il intègre un paramètre obligatoire supplémentaire, le `separatorBuilder`, qui attend une instance de `IndexedWidgetBuilder`. Son implémentation sera réalisée comme dans l'`itemBuilder`, c'est-à-dire en incorporant un `BuildContext` et un `index`. En revanche, là où le retour était un objet `Dismissible`, voire une `ListTile` directement, ici il va être recommandé d'utilise le widget `Divider`.

Avec la même base de code que dans l'exemple précédent, la `ListView` se retrouverait sous cette forme :

```
ListView.separated(
   separatorBuilder: (context, index) => Divider(color:
Colors.cyan,),
   itemCount: _voitures.length,
   itemBuilder: (context, index) {
     final item = _voitures[index];
     return Dismissible( ...
```

Le `separatorBuilder` renvoie un objet `Divider` par l'intermédiaire d'une fonction fléchée. Pour mieux apprécier les traits de séparation, ceux-ci sont colorés en cyan comme le reste de l'application. Visuellement, cela ressemblera à ceci :

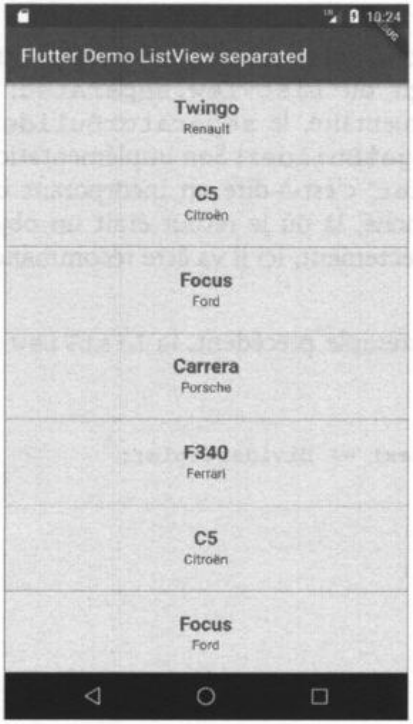


Figure 3 - ListView.separated

© Editions ENI - All rights reserved

2.5 Personnalisation

Si le style des `ListTile` ne convient pas, il est possible de personnaliser la présentation des items.

2.5.1 Card

Dans un premier temps, le désir est d'avoir des `ListTile` qui vont être séparées les unes des autres, cette fois-ci pas seulement par un trait mais en leur donnant la forme de tuiles. Il existe le widget `Card`. Comme son nom le laisse sous-entendre, il ressemble à une carte. Le widget `Card` possède une élévation (`elevation`) qui permet de lui procurer un effet de hauteur. Il est possible de la colorer (`color`). Elle possède une bordure contrôlable grâce à la propriété `shape`.

Par défaut, elle occupe la largeur disponible au sein de son parent. Pour jouer sur sa taille, le paramètre `margin` peut gérer les espacements alentours.

Enfin, `Card` contient un enfant (`child`) qui prendra naturellement une `ListTile`.

Pour reprendre le premier exemple du `ListView.builder`, une instance `Card` est ajoutée en tant qu'enfant au `Dismissible`. Elle prendra elle-même en `child` la `ListTile` :

```
ListView.builder(
   itemCount: _voitures.length,
   itemBuilder: (context, index) {
     final item = _voitures[index];
     return Dismissible(
       key: Key(item.modele),
       background: Container(
         child: Icon(
           Icons.delete,
           size: 40,
           color: Colors.white,
         ),
         color: Colors.cyan,
       ),
       onDismissed: (direction) {
         setState(() {
           _voitures.removeAt(index);
```

```
        });
        Scaffold.of(context).showSnackBar(SnackBar(
            content: Text('Voiture ${item.modele} supprimée !')));
      },
      child: Card( //********* Insertion de la Card *********
        child: ListTile(
          title: Text(
            '${item.modele}',
            textAlign: TextAlign.center,
            style: TextStyle(
                fontWeight: FontWeight.bold,
                color: Colors.cyan,
                fontSize: 20),
          ),
          subtitle: Text(
            '${item.marque}',
            textAlign: TextAlign.center,
          ),
          onTap: () {
            showDialog(
                context: context,
                builder: (BuildContext context) {
                  return AlertDialog(
                    backgroundColor: Colors.cyan,
                    title: Text(
                      'Photo de la voiture ${item.modele}',
                      style: TextStyle(color: Colors.white),
                    ),
                    content: item.image,
                  );
                });
          },
        ),
      ),
    );
  }),
```

© Editions ENI - All rights reserved

Le reste du code est inchangé. L'instance de `Card` n'est pas surchargée. Elle prend donc son style par défaut. L'apparence de l'écran vue sur la figure 1 évolue pour ressembler à cela :

Figure 4 - Mise en place de Card

2.5.2 Inkwell

L'avantage d'utiliser `ListTile`, en dehors des aspects graphiques (`leading`, `trailing`, `title`, `subtitle`), est l'accès à `onTap` (et `onLongPress`). Il serait possible de travailler à partir d'autres widgets, mais ils ne comportent pas tous la gestion de ces événements d'une part, et ils n'offrent pas tous les caractéristiques esthétiques utiles pour l'affichage d'items d'une liste d'autre part.

Afin de se débarrasser de `ListTile` et de pousser la personnalisation plus loin, il est crucial d'évoquer le widget `InkWell`. Ce widget correspond à un rectangle qui réagit au toucher. À ce titre, il possède une grande palette d'attributs consacrés à la gestion d'évènement :

- `onDoubleTap` : appel sur un double appui.
- `onFocusChange` : appel quand le focus change.
- `onHighlightChanged` : appel quand le widget commence (ou arrête) le *highlight*. C'est-à-dire lorsqu'il s'illumine, se surligne.
- `onHover` : appel au survol du widget.
- `onLongPress` : appel sur un appui long.
- `onTap` : appel sur un appui.
- `onTapCancel` : appel sur un appui annulé.
- `onTapDown` : appel sur un appui.

Remarque

La différence entre `onTap`, `onTapDown` et `onTapCancel` est subtile. Sur un geste d'appui, le moment où le doigt entre en contact avec le widget est le `TapDown`. À l'instant où le doigt se retire, le `TapUp` se déclenche. Le fait de combiner les deux, c'est-à-dire d'appuyer puis de retirer son doigt correspond au `Tap`. Si pendant le toucher, le doigt change de zone et sort du widget, l'appui est considéré annulé et c'est `TapCancel` qui s'exécute.

En raison de sa sensibilité au toucher, ce widget comporte un aspect graphique intéressant avec la gestion d'un *highlight* et d'un *splash*. Il s'agit d'un effet ressemblant à une propagation d'ondes partant de la zone touchée jusqu'aux abords du widget. La «zone d'ondes» est le *splash* et la *highlight* la zone sans propagation.

© Editions ENI - All rights reserved

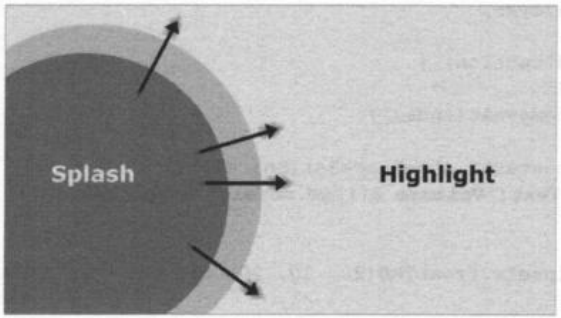


Figure 5 - Splash et highlight

Il est possible de configurer les couleurs des deux éléments grâce aux propriétés `splashColor` et `highlightColor`.

Pour mettre en lumière cet outil et par là-même pousser un peu en termes de personnalisation, l'exemple des voitures va être retravaillé.

La base reste la même et un `ListView.builder` est employé. Ce dernier renvoie un widget `Dismissible` pour permettre le glissement des items (*swipe*) avec en guise d'objet `child` une `Card`.

Voici le code de ce squelette :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.cyan,
   ),
   body: ListView.builder(
       itemCount: _voitures.length,
       itemBuilder: (context, index) {
         final item = _voitures[index];
         return Dismissible(
           key: Key(item.modele),
           background: Container(
             child: Icon(
               Icons.delete,
               size: 40,
               color: Colors.white,
             ),
```

```
            color: Colors.cyan,
          ),
          onDismissed: (direction) {
            setState(() {
              _voitures.removeAt(index);
            });
            Scaffold.of(context).showSnackBar(SnackBar(
                content: Text('Voiture ${item.modele} supprimée !')));
          },
          child: Card(
            margin: EdgeInsets.fromLTRB(20, 10, 20, 5),
            child: ...
```

Concernant l'enfant (`child`) de la `Card`, le choix va se porter sur un `Container`. Ce widget est facile à travailler graphiquement grâce à son paramètre `decoration`. Un style unique va pouvoir être mis en place. Une ombre est placée via `bowShadow` ainsi qu'un dégradé grâce à la propriété `gradient`. En l'occurrence, un dégradé linéaire (`LinearGradient`) est choisi en gardant la couleur déjà utilisée, à savoir cyan.

```
child: Container(
 height: 50,
 decoration: BoxDecoration(
   boxShadow: [
     BoxShadow(
         color: Colors.grey,
         blurRadius: 10,
         offset: Offset(-15.0, 0.0))
   ],
   gradient: LinearGradient(
       colors: [Colors.cyan[50], Colors.cyan[600]],
       begin: Alignment.centerLeft,
       end: Alignment.centerRight),
 ),
 child: ...
```

Il reste à incorporer l'`InkWell`. Dans le lot d'événements utilisables, l'application va se focaliser sur l'appui classique appelé par `onTap`. Ici, le code ne va pas varier de ce qui était fait précédemment. Un pop-up montrant une image de voiture va s'ouvrir.

© Editions ENI - All rights reserved

En revanche, l'`InkWell` prendra pour enfant (`child`) une `Row`. Celle-ci sera composée de deux textes présentant respectivement le modèle et la marque de la voiture.

```
InkWell(
 child: Row(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
   children: <Widget>[
     Text(
       '${item.modele}',
       style: TextStyle(
           color: Colors.cyan[700],
           fontWeight: FontWeight.bold,
           fontSize: 30),
     ),
     Text(
       '${item.marque}',
       style: TextStyle(color: Colors.white),
     ),
   ],
 ),
 onTap: () {
   showDialog(
       context: context,
       builder: (BuildContext context) {
         return AlertDialog(
           backgroundColor: Colors.cyan,
           title: Text(
             'Photo de la voiture ${item.modele}',
             style: TextStyle(color: Colors.white),
           ),
           content: item.image,
         );
       });
 },
),
```

Graphiquement, il est donc possible d'aller très loin avec les nombreuses fonctionnalités qu'offre Flutter. Concernant l'application de l'exemple, le rendu final est le suivant :

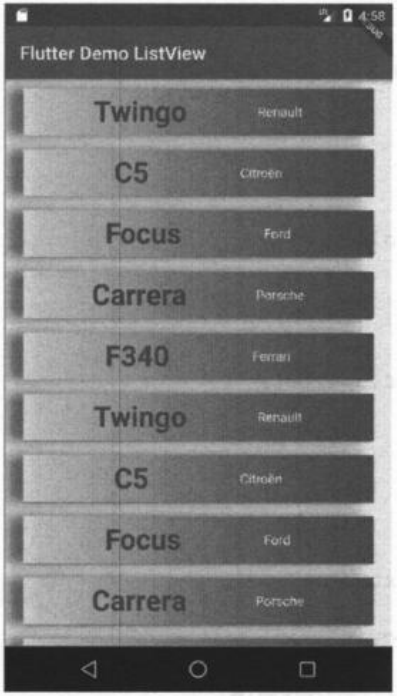


Figure 6 - Personnalisation des items d'une ListView

© Editions ENI - All rights reserved

3. GridView

Les grilles sont une autre forme d'affichage des collections. À la différence de `ListView`, le widget `GridView` autorise l'affichage sous forme de vignettes. L'effet est donc différent, mais dans le cas où des images ou des icônes sont présentes dans la collection, ce type de présentation est plus profitable.

3.1 GridView.builder

Tout comme pour les listes, un constructeur nommé `GridView.builder` permet de mettre en place facilement le widget. Il faut toutefois noter la présence d'un paramètre à prendre en considération, `gridDelegate`. Il permet de choisir notamment le nombre de colonnes sur lesquelles les vignettes vont se répartir. Dans le code qui va servir d'exemple, toujours avec les voitures, le paramètre `gridDelegate` recevra une instance de `SliverGridDelegateWithFixedCrossAxisCount` qui permet de définir un nombre fixe de colonnes. Dans cas précis, il sera défini à 3. Pour le reste, le paramètre `itemCount` ainsi que `itemBuilder` seront présents et travailleront de la même façon que dans les autres cas pratiques.

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.cyan,
   ),
   body: GridView.builder(
       gridDelegate:
           SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 3),
       itemCount: _voitures.length,
       itemBuilder: (context, index) {
         final item = _voitures[index];
         return ...
```

3.2 GridTileBar

Concernant le retour de l'`itemBuilder`, il existe un outil déjà prêt pour cet usage. Il s'agit du `GridTileBar`. Il fonctionne exactement comme le widget `ListTile`. Les paramètres `title`, `subtitle`, `leading` et `trailing` sont présents. Il est personnalisable grâce à `backgroundColor`. En revanche, il ne propose pas d'événement `onTap`.

Dans cet exemple, il est envisageable d'utiliser un `StatelessWidget` puisqu'aucune gestion d'état ne va être gérée.

Son implémentation est donc assez aisée :

```
itemBuilder: (context, index) {
  final item = _voitures[index];
  return GridTileBar(
    backgroundColor: Colors.cyan,
    subtitle: Text(
      '${item.modele}',
      style: TextStyle(
        color: Colors.white,
        fontWeight: FontWeight.bold,
        fontSize: 20),
      textAlign: TextAlign.end,
    ),
    title: Container(
      width: MediaQuery.of(context).size.width * 0.25,
      child: item.image,
    ),
  );
}
```

Dans le contexte qui est celui d'une vignette, le choix est fait d'incorporer comme titre directement l'image de la voiture. Ce choix est aussi conditionné par le fait qu'en l'absence d'événement `onTap`, il sera impossible de faire apparaître un pop-up comme c'était le cas dans les `ListView`.

© Editions ENI - All rights reserved

Voici ce que donne l'application :

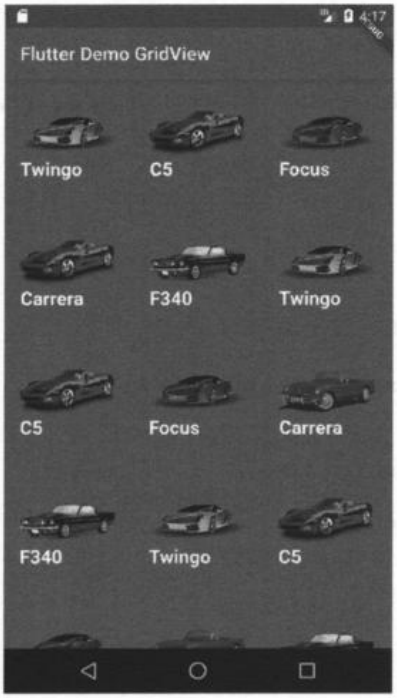


Figure 7 - GridView avec GridTileBar

3.3 Personnalisation

La présentation de l'application telle qu'elle est illustrée à la figure 7 n'est pas optimale. Le meilleur outil dans cette configuration reste le widget `Card`. Il va offrir plus de souplesse et permettra de personnaliser facilement le rendu. Ce widget combiné avec un `Container` donne la possibilité de placer les éléments avec beaucoup de précision.

En restant toujours dans la même optique d'afficher des voitures, le code suivant pourrait être utilisé :

```
GridView.builder(
   gridDelegate:
   SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 2),
   itemCount: _voitures.length,
   itemBuilder: (context, index) {
     final item = _voitures[index];
     return Card(
       color: Colors.cyan,
       elevation: 3.0,
       child: Container(
         padding: EdgeInsets.fromLTRB(15, 0, 15, 0),
         child: Column(
           mainAxisAlignment: MainAxisAlignment.spaceEvenly,
           children: <Widget>[
             item.image,
             Divider(color: Colors.white,),
             Text('${item.modele}', style: TextStyle(color:
Colors.white, fontWeight: FontWeight.bold, fontSize: 20),),
             Text('${item.marque}', style: TextStyle(color:
Colors.white))
           ],
         ),
       ),
     );
   }),
```

La logique vise à obtenir, pour chaque vignette, une représentation verticale comprenant l'image puis le modèle et enfin la marque. L'image et la zone de texte seront séparées par une ligne horizontale (`Divider`). Le texte du modèle est un peu plus grand, ce qui permet de le démarquer.

Il faut noter aussi que, cette fois, la grille va apparaître sur deux colonnes.

© Editions ENI - All rights reserved

Voici le rendu de l'application :

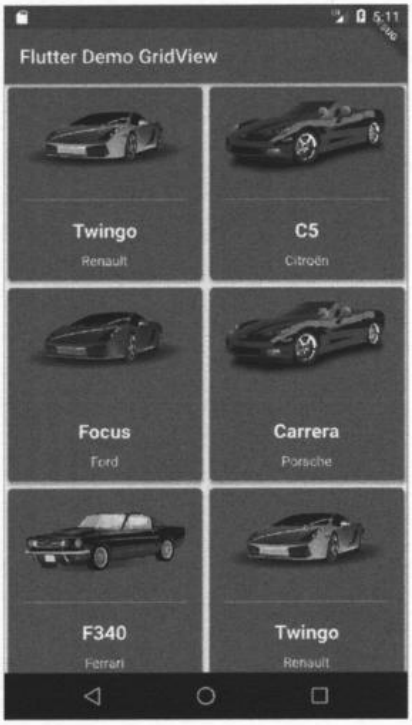
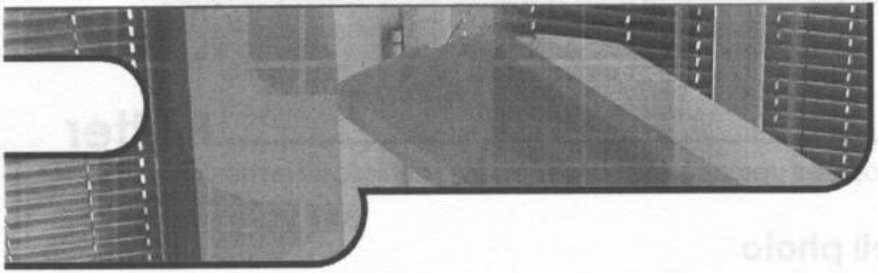


Figure 8 - GridView avec Card



Chapitre 11
API

1. Introduction

En fonction des projets d'applications, il est parfois utile de recourir au hardware de l'appareil, c'est-à-dire sa partie matérielle.

Communément, les deux outils qui servent le plus restent l'appareil photo et le GPS. Ces éléments ne sont pas accessibles directement. L'utilisation de dépendances permettant de les mettre en œuvre facilement est donc indispensable.

Dans ce chapitre, deux exemples concrets sont donnés pour recourir à l'utilisation de l'appareil photo et du GPS.

Pour finir, un point est fait sur les packages disponibles aujourd'hui qui viennent compléter cette offre afin d'avoir accès à d'autres fonctionnalités comme le gyroscope, l'accéléromètre, la batterie…

2. Appareil photo

L'utilisation de l'appareil photo n'est pas directement intégrée dans le SDK. Il convient de faire appel à un package spécialement dédié à cette tâche. Il en existe plusieurs, mais le plus fiable à ce jour est développé par l'équipe Flutter. Il s'agit d'**image_picker**. Ce package possède un double rôle puisqu'il permet de prendre des photos et donc d'utiliser le hardware, mais il peut aussi lire des fichiers images présents dans le téléphone.

À ce jour, il possède la note maximale de 100.

Voici l'URL permettant d'y accéder : https://pub.dev/packages/image_picker

2.1 Installation du package

La première étape consiste, comme à chaque fois, à modifier le fichier *pubspec.yaml* sous `dependencies` :

```
image_picker: ^0.6.4
```

Attention, il ne faut pas oublier de cliquer sur **Packages get** pour récupérer cette dépendance et pouvoir l'utiliser par la suite.

2.2 Mise à jour des permissions

Précédemment, nous avons évoqué très brièvement les permissions. Elles sont gérées dans des fichiers spécifiques pour chaque OS. Pour Android, le fichier *AndroidManifest.xml* joue ce rôle tandis que pour iOS, le fichier se nomme *Info.plist*.

Voici l'emplacement de ces fichiers :

- *AndroidManifest.xml* :
  Racine du projet -> android -> app -> src -> main
- *Info.plist* :
  Racine du projet -> ios -> Runner

© Editions ENI - All rights reserved

Dans le cadre de l'utilisation du package **image_picker**, il n'y a pas de permission à ajouter pour Android. En revanche, pour iOS, il va falloir les mettre en place. Voici ce qu'il faut ajouter à *Info.plist* :

```
<dict>
    <key>NSPhotoLibraryUsageDescription</key>
    <string>Utilisation de la librairie de photos</string>
    <key>NSCameraUsageDescription</key>
    <string>Utilisation de la camera</string>
...
```

La première permission est `NSPhotoLibraryUsageDescription`. Elle donne la possibilité d'accéder à la lecture des images dans l'appareil.

Enfin, `NSCameraUsageDescription` permet de prendre le contrôle de la caméra et donc de prendre de nouvelles photographies.

Maintenant que la configuration est terminée, il est possible de s'attaquer au code.

2.3 Le code du programme

Cette application va se reposer sur la classe `StatefulWidget`. Elle devra gérer les états afin de mettre à jour l'affichage. Le code de départ est donc classique et déjà vu :

```
import 'dart:io';
import 'package:flutter/material.dart';
import 'package:image_picker/image_picker.dart';

void main() {
 runApp(MyApp());
}

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     home: MyHomePage(title: 'Demo camera'),
   );
 }
```

```
}
class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
```

Il faut malgré tout noter la présence, dans les importations, du package `image_picker` chargé précédemment, ainsi que de `dart:io` qui sera utile pour la manipulation de fichier.

À ce propos, dans `_MyHomePageState`, il convient de déclarer deux variables. La première, justement, est de type File. Elle servira à contenir l'image, peu importe qu'elle provienne d'une nouvelle photo prise avec la caméra ou d'un fichier déjà stocké sur l'appareil.

La seconde va permettre de récupérer les erreurs éventuelles. Son type est `dynamic`.

```
class _MyHomePageState extends State<MyHomePage> {
 File _imageFile;
 dynamic _pickImageError;
 ...
```

La partie graphique va être assez simple. Elle sera composée d'une `AppBar` afin d'avoir un titre. L'image qui vient d'être prise par l'appareil photo ou celle sélectionnée dans les fichiers du smartphone devra s'afficher dans le `body`. Enfin, pour pouvoir justement prendre une photo ou choisir une image, deux boutons d'action flottants sont intégrés en bas de l'écran.

Voici le code de cette interface :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.green,
```

© Editions ENI - All rights reserved

```
    ),
    body: Center(
      child: _visualiserImage(),
    ),
    floatingActionButton: Row(
      mainAxisAlignment: MainAxisAlignment.end,
      children: <Widget>[
        Padding(
          padding: const EdgeInsets.all(10.0),
          child: FloatingActionButton(
            backgroundColor: Colors.green,
            onPressed: () {
              _onImageButtonPressed(ImageSource.gallery);
            },
            child: const Icon(Icons.photo_library),
          ),
        ),
        Padding(
          padding: const EdgeInsets.all(10.0),
          child: FloatingActionButton(
            backgroundColor: Colors.green[800],
            onPressed: () {
              _onImageButtonPressed(ImageSource.camera);
            },
            child: const Icon(Icons.photo_camera),
          ),
        ),
      ],
    ),
  );
}
```

Les deux boutons d'action flottants sont rangés dans une Row pour les aligner horizontalement. Ils appellent tous les deux la même méthode `_onImageButtonPressed`. La différence se situe dans le paramètre qui lui est passé. Il indique la source de l'image, à savoir l'appareil photo ou la bibliothèque du téléphone.

Le body fait appel à une méthode nommée `_visualiserImage`. Elle va, comme son nom l'indique, servir à afficher l'image.

Commençons par détailler l'action des boutons. Le package offre simplement une méthode intitulée `pickImage` qui attend la source de l'image. C'est sur elle que le code va s'appuyer. Ce sera également l'occasion de mettre à jour l'interface et de rappeler l'exécution de la méthode `_visualiserImage`.

Cette étape de récupération d'une image reste toujours risquée. Il convient de prendre des précautions et de la placer dans un `try/catch` afin d'intercepter les éventuelles erreurs. Elles seront affectées à la variable prévue à cet effet `_pickImageError`.

```
void _onImageButtonPressed(ImageSource source) async {
 try {
   _imageFile = await ImagePicker.pickImage(
     source: source,
   );
   setState(() {});
 } catch (e) {
   _pickImageError = e;
 }
}
```

L'image récupérée est affectée à la variable `_imageFile`. L'appel à `ImagePicker.pickImage` est logiquement asynchrone. Le paramètre `source` permet de préciser d'où vient l'image (appareil photo ou dossier des images).

L'appel à `setState` sert uniquement à redéclencher le `build`. La méthode `_visualiserImage` est donc exécutée de nouveau.

Dans le cas d'une levée d'exception, le bloc `catch` prévoit que la variable `_pickImageError` récupère l'erreur.

© Editions ENI - All rights reserved

Voyons la méthode `_visualiserImage` :

```
Widget _visualiserImage() {
 if (_imageFile != null) {
   return Image.file(_imageFile);
 } else if (_pickImageError != null) {
   return Text(
     'Erreur de récupération d\'image: $_pickImageError',
     textAlign: TextAlign.center,
   );
 } else {
   return const Text(
     'Aucune image',
     textAlign: TextAlign.center,
   );
 }
}
```

Cette méthode renvoie un `Widget`. Elle doit, en effet, alimenter l'élément `body` de l'application. Trois cas sont prévus. Le premier vérifie si la variable `_imageFile` contient une donnée. Si oui, l'image est renvoyée grâce au constructeur nommé `Image.file`.

Sinon, si la variable `_pickImageError` est différente de `null`, c'est-à-dire qu'une erreur est survenue pendant l'exécution de `_onImageButtonPressed`, cette dernière est affichée dans un `Text`.

Enfin, si aucune image ni aucune erreur ne sont présentes, par défaut un objet `Text` prévient qu'il n'y a aucune image.

2.4 Rendu

Figure 1 - image-picker

2.5 Vidéo

La vidéo n'est pas directement gérée avec le package **image_picker**. Mais il existe un équivalent pour la capture de vidéo. Le package en question se nomme **video_player**. Il est accessible à cette adresse :
https://pub.dev/packages/video_player

© Editions ENI - All rights reserved

3. GPS

L'utilisation du GPS se passe d'une manière proche de celle développée ci-dessus pour les images.

Il faut recourir à un package permettant d'accéder à la puce GPS. En l'occurrence, il en existe plusieurs qui sont plus ou moins complets.

Parmi eux, le package de l'équipe Flutter, **gps**, en version 0.1.1 à ce jour. Vraisemblablement, il ne s'agit pas du meilleur aujourd'hui. Il obtient une note de seulement 88 sur 100. Il s'avère assez peu complet, mais reste très accessible.

Il est disponible ici : https://pub.dev/packages/gps

La communauté se tourne plus vers **geolocator** ou **location** qui possèdent tous les deux une note de 100.

Geolocator est disponible ici : https://pub.dev/packages/geolocator

Location est disponible ici : https://pub.dev/packages/location

Pour l'exemple qui va suivre le package **location** va être utilisé. Il est plus simple d'emploi et convient mieux au débutant.

3.1 Installation du package

Afin d'avoir accès à l'ensemble des éléments du package location, il convient de le déclarer dans le fichier *pubspec.yaml* :

```
dependencies:
  location: ^3.0.2
```

Après avoir sauvegardé cette modification, il suffit maintenant de cliquer sur **Packages get** pour récupérer les sources.

3.2 Mises à jour des permissions

L'utilisation de la puce GPS afin de déterminer la position de l'utilisateur requiert des permissions. Elles sont à définir dans les fichiers *AndroidManifest.xml* et *Info.plist*.

Voici l'emplacement de ces fichiers :

– *AndroidManifest.xml* :
Racine du projet -> android -> app -> src -> main

– *Info.plist* :
Racine du projet -> ios -> Runner

Les permissions à ajouter sont les suivantes :

– Pour Android :

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

– Pour iOS :

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>Utilisation de la position</string>
<key>NSLocationAlwaysUsageDescription</key>
<string>Utilisation de la position</string>
```

3.3 Le code du programme

L'objet du programme est simple. Il s'agit d'obtenir la latitude et la longitude et de les afficher sur l'écran. `MyHomePage` va étendre `StatefulWidget`. Non pas pour travailler avec `setState` mais pour bénéficier du cycle de vie, et notamment de la méthode `initState`. Au lancement du programme, la position va être récupérée.

© Editions ENI - All rights reserved

La base du programme est la suivante :

```
import 'package:flutter/material.dart';
import 'package:location/location.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo GPS'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);
 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
...
```

Pour utiliser le package, un `import` de **location** est réalisé. Deux variables vont ensuite être utiles. La première est une instance de `Location`, la seconde de `LocationData`.

```
Location _location = Location();
LocationData _locationData;
```

Dans la méthode `initState`, l'enjeu va être maintenant d'obtenir la position :

```
@override
void initState() {
 super.initState();
```

```
  _trouverPosition();
}

Future<void> _trouverPosition() async {
 try {
   _locationData = await _location.getLocation();
 } catch (e) {
   print('Erreur lors de la récupération de la position : $e');
 }
}
```

`InitState` appelle une méthode intitulée `_trouverPosition`. Celle-ci cherche à récupérer la position exacte grâce à la méthode `getLocation`. Elle est asynchrone, le traitement peut prendre du temps. Le tout est exécuté dans un `try/catch` pour plus de prudence. Une fois cette exécution terminée, la variable `_locationData` contiendra les informations de positionnement de l'utilisateur.

À partir de là, il ne reste plus qu'à afficher ces informations. L'interface graphique est très simple :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
   ),
   body: Center(
     child: Column(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'Position actuelle : ',
         ),
         Text(
           _locationData == null
               ? 'En attente de position'
               : 'Longitude : ${_locationData.longitude}',
           style: Theme.of(context).textTheme.display1,
         ),
         Text(
           _locationData == null
               ? 'En attente de position'
```

© Editions ENI - All rights reserved

```
                    : 'Latitude : ${_locationData.latitude}',
                style: Theme.of(context).textTheme.display1,
              ),
            ],
          ),
        ),
      );
    }
```

Dans l'hypothèse où `_locationData` serait `null`, un texte de remplacement est prévu. Si ce n'est pas le cas, alors les valeurs des attributs `longitude` et `latitude` sont appelées.

3.4 Rendu

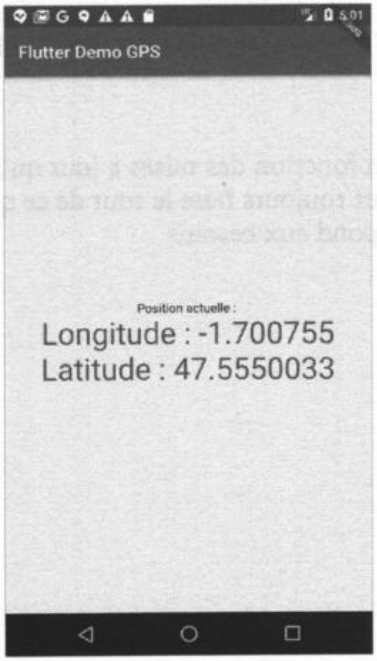


Figure 2 - Géolocalisation avec Location

4. Autres API

Il existe de nombreux packages et ressources permettant de contrôler le hardware des appareils. Sans vouloir en faire une liste exhaustive, il est intéressant de noter la présence de quelques dépendances utiles :

- Pour l'accéléromètre et le gyroscope : **sensors** de l'équipe Flutter :

  https://pub.dev/packages/sensors
- Pour contrôler l'état du réseau mobile et du Wi-Fi ou obtenir des informations dessus : **connectivity** de l'équipe Flutter :

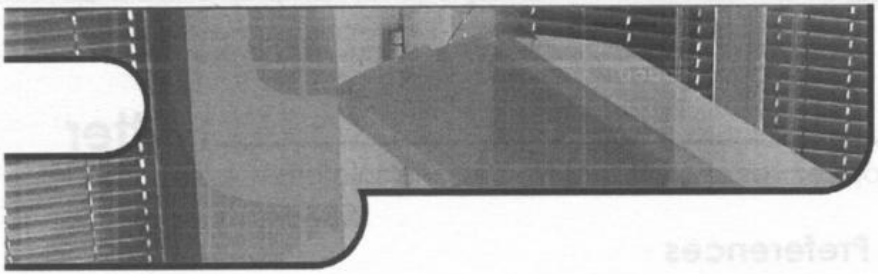
  https://pub.dev/packages/connectivity
- Pour obtenir des informations sur l'appareil (modèle, marque, version d'OS...) : **device_info** de l'équipe Flutter :

  https://pub.dev/packages/device_info
- Pour obtenir des informations concernant la batterie : **battery** de l'équipe Flutter :

  https://pub.dev/packages/battery

Ces packages évoluent plus ou moins vite en fonction des mises à jour qu'ils subissent. Avant de choisir un package, il faut toujours faire le tour de ce qui est proposé et bien vérifier que le package répond aux besoins.

© Editions ENI - All rights reserved



Chapitre 12
Données

1. Introduction

L'utilisation de données est une pratique indispensable. Il est nécessaire de comprendre comment sauvegarder ces données et pouvoir les réutiliser. Dans ce chapitre, plusieurs voies vont être abordées.

D'abord, nous verrons le système de stockage appelé Shared Preferences. Il s'agit d'un outil très pratique et surtout facile à mettre en œuvre. En revanche, il n'offre pas de garanties suffisantes pour le stockage de données plus sensibles ou qui ont vocation à être gardées plus longtemps. Pour cela, nous aborderons la base de données embarquée SQLite.

Enfin, puisque le modèle aujourd'hui s'y prête, nous étudierons l'exploitation de données au format JSON au travers d'une API.

2. Shared Preferences

2.1 Introduction

Les Shared Preferences sont un bon moyen d'enregistrer des données avec Flutter. Cette pratique découle de la fonctionnalité du même nom disponible sous Android et de NSUserDefaults sous iOS.

Grâce à ce système, les données peuvent être rendues persistantes sur l'appareil de l'utilisateur de manière asynchrone. Toutefois, cette méthode de persistance n'est pas garantie. Il convient donc d'éviter de sauvegarder des données importantes par ce biais.

2.2 Récupération des dépendances

L'utilisation de Shared Preferences nécessite le package **shared_preferences**. Sa déclaration s'opère en renseignant le fichier *pubspec.yaml*:

```
dependencies:
  shared_preferences: ^0.5.6+3
```

Une fois déclarée, la dépendance sera récupérée en cliquant sur **Packages get**.

Remarque

Ce package appartient au programme Flutter Favorite. Il est donc recommandé par l'équipe Flutter.

2.3 Création de l'application

Le dessein de l'application est d'enregistrer les données d'une personne dans les Shared Preferences et de pouvoir les récupérer pour les afficher à l'écran. L'application permettra également de supprimer ces données au besoin.

© Editions ENI - All rights reserved

2.3.1 Classe Personne

Dans un nouveau projet Flutter, un fichier nommé *personne.dart* contenant une classe intitulée `Personne` est créé dans le dossier *lib*. Elle comprend trois attributs (nom, prénom et âge), un constructeur et les accesseurs.

```
class Personne {
 String _nom;
 String _prenom;
 String _age;

 Personne(this._nom, this._prenom, this._age);

 String get age => _age;

 set age(String value) {
   _age = value;
 }

 String get prenom => _prenom;

 set prenom(String value) {
   _prenom = value;
 }

 String get nom => _nom;

 set nom(String value) {
   _nom = value;
 }
}
```

Cette classe servira de support à la création d'une nouvelle personne qui sera utilisée tout au long du programme.

2.3.2 main.dart

Dans le fichier *main.dart*, `MyHomePage` étend `StatefulWidget` afin de bénéficier du cycle de vie de ce widget et de la mise à jour du `state`.

```
import 'package:flutter/material.dart';
import 'package:fluttersharedpreferences/Personne.dart';
import 'package:shared_preferences/shared_preferences.dart';
```

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
     theme: ThemeData(
       primarySwatch: Colors.blue,
     ),
     home: MyHomePage(title: 'Flutter Demo Shared Preferences'),
   );
 }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
...
```

Dans la classe `_MyHomePageState`, il convient de déclarer trois variables. La première est une instance de `Personne`. Elle va servir de support à l'enregistrement des données dans les Shared Preferences. La seconde est une autre instance de `Personne`. Elle va se voir affecter les données récupérées un peu plus tard. Enfin, pour que tout ceci puisse fonctionner, il faut récupérer une instance de `SharedPreferences`. Cette opération s'effectue par l'appel asynchrone de la méthode `getInstance`.

```
final Personne personneEnregistree = Personne("DOE", "John", "35");
Personne personneRecuperee;
Future<SharedPreferences> _prefs = SharedPreferences.getInstance();
```

L'interface graphique est composée de deux zones principales. La première va afficher les données de la personne déjà créée. La seconde affichera les données stockées. Théoriquement, si tout se passe bien, elles seront identiques dans les deux endroits.

© Editions ENI - All rights reserved

En dessous, trois boutons sont disposés. Le premier sert à enregistrer les données. Le second à les lire et le dernier à les effacer. Chacun sera relié à une méthode chargée d'une de ces responsabilités.

Le code commence par intégrer une `AppBar` :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
   ),
```

Puis le `body` comporte les traditionnels `Center` et `Colum` :

```
body: Center(
 child: Column(
   mainAxisAlignment: MainAxisAlignment.spaceAround,
   children: <Widget>[
```

Dans la liste des enfants (`children`) de la colonne, une icône symbolisant la connexion à un compte est placée en guise de widget de départ. Elle n'a qu'un rôle purement esthétique :

```
Icon(
 Icons.account_circle,
 size: 80.0,
 color: Colors.blue,
),
```

Au-dessous, le premier cadre d'information est créé. Il s'agit d'un `Container` décoré d'un fond gris et de bords arrondis. À l'intérieur se trouvent trois lignes indiquant respectivement le nom, le prénom et l'âge contenus dans la variable `personneEnregistree`.

```
Text('Données soumises :'),
Container(
 height: MediaQuery.of(context).size.height * 0.25,
 width: MediaQuery.of(context).size.width * 0.8,
 decoration: BoxDecoration(
     color: Colors.grey,
     shape: BoxShape.rectangle,
     borderRadius: BorderRadius.circular(20.0)),
 child: Column(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
```

```
    children: <Widget>[
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Nom : ',
            style: TextStyle(color: Colors.white),
          ),
          Text('${personneEnregistree.nom}',
              style: TextStyle(
                  fontSize: 30.0,
                  color: Colors.white,
                  fontWeight: FontWeight.bold)),
        ],
      ),
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Prénom : ',
            style: TextStyle(color: Colors.white),
          ),
          Text('${personneEnregistree.prenom}',
              style: TextStyle(
                  fontSize: 30.0,
                  color: Colors.white,
                  fontWeight: FontWeight.bold)),
        ],
      ),
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Age : ',
            style: TextStyle(color: Colors.white),
          ),
          Text('${personneEnregistree.age}',
              style: TextStyle(
                  fontSize: 30.0,
                  color: Colors.white,
                  fontWeight: FontWeight.bold)),
        ],
      ),
    ],
  ),
),
```

© Editions ENI - All rights reserved

Juste après, une deuxième zone similaire est mise en place. Seule différence, les données affichées ne seront plus celles de `personneEnregistree` mais celles de `personneRecuperee` si elles existent, ou un message de substitution dans le cas inverse.

```
Text('Données récupérées :'),
Container(
 height: MediaQuery.of(context).size.height * 0.25,
 width: MediaQuery.of(context).size.width * 0.8,
 decoration: BoxDecoration(
     color: Colors.blue,
     shape: BoxShape.rectangle,
     borderRadius: BorderRadius.circular(20.0)),
 child: Column(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
   children: <Widget>[
     Row(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'Nom : ',
           style: TextStyle(color: Colors.white),
         ),
         personneRecuperee != null
             ? Text('${personneRecuperee.nom}',
                 style: TextStyle(
                     fontSize: 30.0,
                     color: Colors.white,
                     fontWeight: FontWeight.bold))
             : Text(
                 'Aucune donnée',
                 style: TextStyle(color: Colors.white),
               )
       ],
     ),
     Row(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'Prénom : ',
           style: TextStyle(color: Colors.white),
         ),
         personneRecuperee != null
```

```
            ? Text('${personneRecuperee.prenom}',
                style: TextStyle(
                    fontSize: 30.0,
                    color: Colors.white,
                    fontWeight: FontWeight.bold))
            : Text(
                'Aucune donnée',
                style: TextStyle(color: Colors.white),
              )
        ],
      ),
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Age : ',
            style: TextStyle(color: Colors.white),
          ),
          personneRecuperee != null
            ? Text('${personneRecuperee.age}',
                style: TextStyle(
                    fontSize: 30.0,
                    color: Colors.white,
                    fontWeight: FontWeight.bold))
            : Text(
                'Aucune donnée',
                style: TextStyle(color: Colors.white),
              )
        ],
      ),
    ],
  ),
),
```

Pour finir, une ligne séparatrice (`Divider`) est insérée pour délimiter la zone réservée aux boutons. Au nombre de trois, ils appelleront chacun dans la propriété `onPressed` une méthode qui leur est propre. Pour alléger l'affichage, ils sont placés en ligne (figure 1) :

```
Divider(),
Row(
 mainAxisAlignment: MainAxisAlignment.spaceEvenly,
 children: <Widget>[
   FlatButton(
```

© Editions ENI - All rights reserved

```
        onPressed: enregistrer,
        color: Colors.green,
        textColor: Colors.white,
        child: Text('Enregistrer')),
    FlatButton(
        onPressed: recuperer,
        color: Colors.blue,
        textColor: Colors.white,
        child: Text('Lire les données')),
    FlatButton(
        onPressed: supprimer,
        color: Colors.red,
        textColor: Colors.white,
        child: Icon(
          Icons.delete,
          color: Colors.white,
        ))
  ],
)
```

Le premier bouton fait référence à une méthode nommée `enregistrer`. Elle sera asynchrone puisque cette action peut prendre du temps. Voici le code qui va être détaillé juste après :

```
Future<void> enregistrer() async {
  final SharedPreferences prefs = await _prefs;
  prefs.setStringList("personne", [
    '${personneEnregistree.nom}',
    '${personneEnregistree.prenom}',
    '${personneEnregistree.age}'
  ]);
  showDialog(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: Text('Enregistrement'),
          content: Text('Les données ont été enregistrées !'),
        );
      });
}
```

La méthode est donc de type `Future`. Elle est, comme cela avait été indiqué un peu plus haut, asynchrone. La première étape est de récupérer une instance de `SharedPreferences`. Pour cela, il est fait appel à la variable `_prefs` déclarée précédemment. Pour rappel, cette déclaration était la suivante :

```
Future<SharedPreferences> _prefs = SharedPreferences.getInstance();
```

Une fois cette instance récupérée, il existe plusieurs méthodes qui permettent l'enregistrement d'informations. Toutefois, elles ne concernent que des types primitifs et non des objets comme c'est le cas avec la `Personne` qui est manipulée dans cet exemple. Dès lors, il convient de trouver une parade. Il pourrait être envisagé de résoudre cette problématique avec JSON. Toutefois, cette notion n'a pas encore été abordée, elle le sera un peu plus loin dans ce chapitre. Une autre solution repose sur la possibilité de transmettre une liste de chaînes de caractères grâce à la méthode `setStringList` associée à une clé qui, dans l'exemple, sera la chaîne « `personne` ». Il va être possible de transmettre, par ce biais, le nom, le prénom et l'âge contenus dans la variable `personneEnregistree`.

Pour confirmer à l'utilisateur que l'enregistrement a eu lieu, une `AlertDialog` s'ouvre à la fin du processus (figure 2).

Ces données sont maintenant stockées dans les Shared Preferences. Le second bouton appelle la méthode `recuperer`. Elle permet de lire la sauvegarde.

```
Future<void> recuperer() async {
 final SharedPreferences prefs = await _prefs;

 if (prefs.getStringList("personne") != null) {
   var liste = prefs.getStringList("personne");
   setState(() {
     personneRecuperee = Personne(liste[0], liste[1], liste[2]);
   });
 }
}
```

La méthode est aussi de type `Future` puisque la récupération de données, tout comme sa sauvegarde, est une opération asynchrone. La première étape consiste encore à récupérer une instance de `SharedPreferences`. Puis, grâce à la méthode `getStringList`, il est possible d'obtenir une liste de chaînes de caractères à partir d'une clé. La clé « `personne` » permet donc d'aller chercher ce qui a été enregistré précédemment.

© Editions ENI - All rights reserved

Une condition est mise en place pour vérifier la nullité de cette requête. Si des données sont bien présentes, un appel à `setState` met à jour l'instance de `personneRecuperee` grâce à son constructeur et l'utilisation en paramètres des différents éléments composant la liste. Sinon, rien ne se passe.

L'avantage de cette précaution est qu'elle permet aussi d'afficher les données au démarrage de l'application si elles existent. À cet effet, un appel à `initState` est intégré :

```
@override
void initState() {
 super.initState();
 recuperer();
}
```

Au démarrage de l'application, la méthode `recuperer` est appelée. Si des données avaient été déjà enregistrées lors d'un précédent lancement du programme, alors elles seront retrouvées et affichées directement (figure 3).

Enfin, dernière fonctionnalité, la suppression. Le troisième bouton prévoit que les données peuvent être effacées des Shared Preferences. La méthode `supprimer` fonctionne comme ceci :

```
Future<void> supprimer() async {
 final SharedPreferences prefs = await _prefs;
 prefs.remove("personne");
 setState(() {
   personneRecuperee = null;
 });
 showDialog(
     context: context,
     builder: (BuildContext context) {
       return AlertDialog(
         title: Text('Suppression'),
         content: Text('Les données ont été supprimées !'),
       );
     });
}
```

Comme pour `enregistrer` et `recuperer`, la première étape consiste à obtenir une instance de `SharedPreferences`. Sa méthode `remove` peut ensuite être appelée. Elle va effacer les données stockées. Pour mettre à jour l'affichage, un appel à `setState` est effectué qui repasse `personneRecuperee` à `null`. Une `AlertDialog` est affichée pour confirmer la suppression (figure 4).

2.4 Rendu de l'application

Au lancement, l'écran apparaît comme ceci :

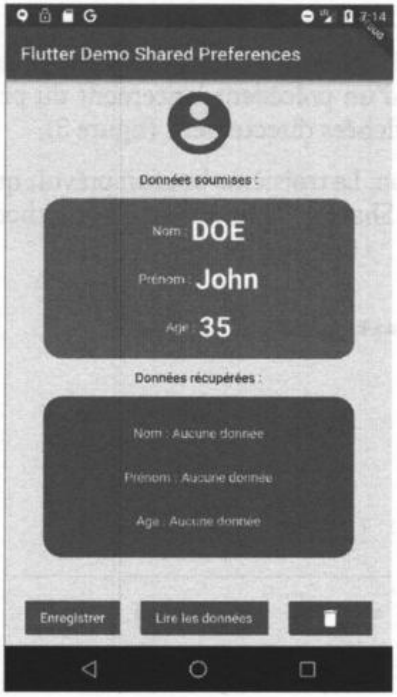


Figure 1 - Shared Preferences, lancement de l'application

© Editions ENI - All rights reserved

Une fois que le bouton **Enregistrer** est appuyé, le pop-up de confirmation de l'enregistrement apparaît :

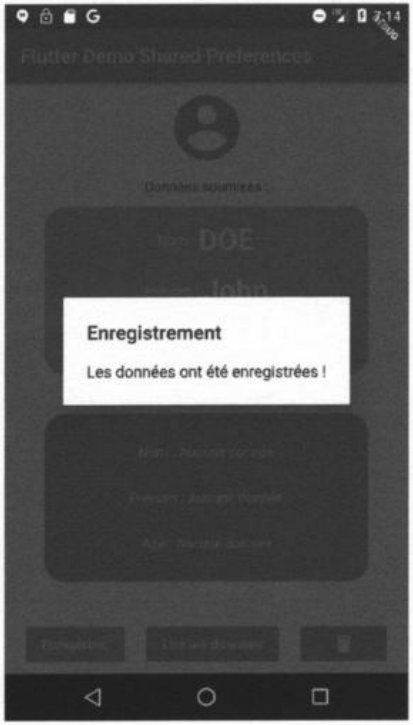


Figure 2 - Enregistrement des données

En appuyant sur le bouton central, les données s'affichent alors dans le second cadre :

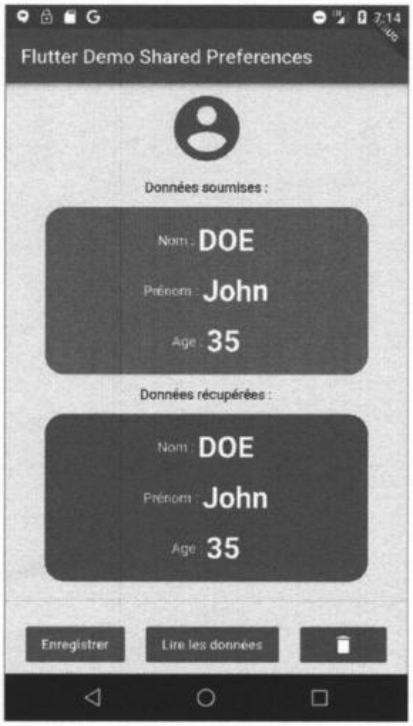


Figure 3 - Affichage des données

© Editions ENI - All rights reserved

Enfin, à l'appui du bouton **Supprimer**, les données sont effacées et un pop-up de confirmation apparaît :

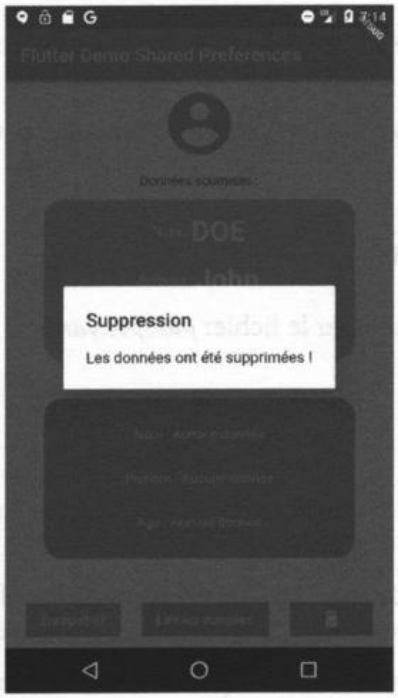


Figure 4 - Suppression des données

3. SQLite

3.1 Introduction

Il existe un autre moyen de conserver des données sur un appareil de manière plus fiable qu'avec les Shared Preferences. Il s'agit d'une base de données interne SQLite. Comme son nom l'indique, SQLite est une base de données de type relationnel avec laquelle le langage SQL s'applique.

Dès lors, les informations pourront être stockées dans une base de données à l'intérieur de tables et réparties dans des colonnes. Cette structure sera plus efficace et plus sûre.

Flutter met à disposition une dépendance qui permet de manipuler la base SQLite : **sqflite**.

En revanche, sans connaissances en langage SQL, le recours à cet outil peut s'avérer être compliqué.

3.2 Récupération des dépendances

Pour récupérer le package **sqflite**, il faut modifier le fichier *pubspec.yaml*.

```
dependencies:
 path_provider: ^1.6.5
 path: ^1.6.4
 sqflite: ^1.3.0
```

Une fois ces trois dépendances ajoutées, il faut terminer en cliquant sur **Packages get**.

Dans le code ci-dessus, deux autres dépendances sont récupérées en complément. Elles seront indispensables dans le cadre de l'exemple montré peu après.

Path_provider permet de trouver les chemins ou emplacements couramment utilisés sur le système de fichiers.

Path est utile dans les opérations permettant la manipulation de chemins ou d'emplacements.

© Editions ENI - All rights reserved

3.3 Création de l'application

Cette application est un peu plus complexe que celle réalisée avec les Shared Preferences. Pour établir un comparatif entre les deux méthodes de persistance des données, c'est la même application qui sera reproduite.

3.3.1 Classe Personne

Dans le dossier *lib*, un fichier nommé *personne.dart* est créé.

L'application s'appuiera donc sur la classe `Personne` qui y est décrite :

```
final String tablePersonne = 'personne';
final String colonneId = 'id';
final String colonneNom = 'nom';
final String colonnePrenom = 'prenom';
final String colonneAge = 'age';

class Personne {
  int id;
  String nom;
  String prenom;
  String age;

  Personne();

  Personne.fromMap(Map<String, dynamic> map) {
    id = map[colonneId];
    nom = map[colonneNom];
    prenom = map[colonnePrenom];
    age = map[colonneAge];
  }

  Map<String, dynamic> toMap() {
    var map = <String, dynamic>{
      colonneNom: nom,
      colonnePrenom: prenom,
      colonneAge: age
    };
    if (id != null) {
      map[colonneId] = id;
    }
```

```
    return map;
  }
}
```

La classe prend une forme un peu différente que celle présentée dans l'exemple précédent. Dans le fichier, pour simplifier le code présenté, deux classes vont être intégrées. La première est donc `Personne`. La seconde sera dévoilée juste après. En tout état de cause, ces deux classes vont utiliser des informations communes qui sont placées en début de fichier. Elles correspondent aux éléments SQL comme le nom de la table, celui des colonnes…

La classe `Personne` est dotée d'un attribut supplémentaire, `id`, qui servira de clé primaire lors du stockage. Comme cette fois l'objectif va être de stocker des valeurs au sein de colonnes, l'utilisation d'une `Map` est plus judicieuse. Elle permet d'associer un couple clé/valeur. La clé correspondra au nom de la colonne.

Pour manipuler cette `Map`, deux méthodes sont prévues. Une première (`toMap`) servira à la lecture des informations contenues dans les attributs et renverra le tout sous la forme de l'objet `map`. La seconde (`Personne.fromMap`) aura pour but d'affecter les valeurs contenues dans l'objet `map` aux attributs de `Personne`. Elle agira donc comme un constructeur nommé.

Noter que dans la méthode `toMap` une condition est posée sur l'`id`. En effet, au moment où l'on voudra récupérer les informations contenues dans une instance, la valeur pour `id` ne sera pas toujours renseignée. Elle le sera si les données ont été insérées dans la base SQLite puisqu'elle sera générée automatiquement par celle-ci.

3.3.2 Classe PersonneProvider

L'autre classe contenue dans ce fichier va être la pierre angulaire de la manipulation de la base de données. Elle est donc assez technique. Elle est nommée `PersonneProvider`.

Pour faciliter son utilisation, une variante du *design pattern* Singleton est mise en place. Cette pratique courante consiste à doter la classe d'un constructeur privé, inaccessible donc de l'extérieur.

© Editions ENI - All rights reserved

```
PersonneProvider._privateConstructor();

static final PersonneProvider instance =
    PersonneProvider._privateConstructor();
```

Ici, le *pattern* n'est que partiellement respecté pour simplifier les choses. L'avantage de cette syntaxe est de pouvoir faire s'exécuter le contrôleur directement ici via l'appel à `instance`. En appelant `instance` ailleurs, ce qui est possible car la variable est `static`, le constructeur va s'exécuter. Ainsi, que ce soit ici, dans `PersonneProvider`, ou dans une autre classe où `instance` est appelée, c'est la même instance qui sera manipulée.

Au-delà, dans le fichier qui contient les classes `Personne` et `PersonneProvider`, plusieurs variables avaient été déclarées de manière globale pour profiter aux deux classes.

Il convient d'en rajouter deux :

```
final String databaseName = 'PersonneDB.db';
final int databaseVersion = 1;
```

Il s'agit du nom de la base de données ainsi que son numéro de version.

Maintenant, plusieurs choses restent à prévoir. Il faut, dans un premier temps, initialiser la base de données. Et pour cela, il nous faut une base :

```
Database _db;
```

Cette base de données est privée. Elle ne pourra donc pas être appelée en dehors de cette classe. Pour travailler avec, il faut mettre en place un *getter* qui, outre le fait de pouvoir récupérer la valeur de la variable _db, gérera son initialisation :

```
Future<Database> get db async {
 if (_db != null) {
   return _db;
 } else {
   _db = await _initDatabase();
   return _db;
 }
}
```

La condition vérifie si la base est initialisée. Si elle l'est, alors la valeur de _db est retournée. Si elle ne l'est pas, la méthode `_initDatabase` est exécutée.

Voici cette méthode qui sera expliquée juste après :

```
_initDatabase() async {
 Directory documentsDirectory = await getApplicationDocumentsDirectory();
 String path = join(documentsDirectory.path, databaseName);
 return await openDatabase(path, version: databaseVersion, onCreate: _open);
}
```

En se servant de **path_provider**, il est possible de récupérer le chemin de notre application :

```
Directory documentsDirectory = await getApplicationDocumentsDirectory();
```

À partir de là, grâce à la fonctionnalité join de **path**, ce chemin est complété par le nom de la base de données :

```
String path = join(documentsDirectory.path, databaseName);
```

Il ne reste plus qu'à exécuter la méthode `openDatabase` qui prend trois paramètres : le chemin de la base (`path`), la version de la base de données et enfin une méthode pour sa création (`onCreate`). Cette dernière méthode est, dans l'exemple, `_open`.

```
Future _open(Database db, int version) async {
 await db.execute('''
   create table $tablePersonne (
   $colonneId integer primary key autoincrement,
   $colonneNom text,
   $colonnePrenom text,
   $colonneAge text)
   ''');
}
```

La méthode `_open` crée la table `personne` dans la base de données. Celle-ci comportera quatre colonnes. La première, pour l'identifiant (`id`), est la clé primaire et sera auto-incrémentée. Les autres sont simplement du texte.

La configuration est presque terminée. Puisque l'ouverture de la base est réalisée, il faut prévoir l'inverse, c'est-à-dire la fermeture. Une méthode intitulée `close` est ajoutée :

```
Future close() async => _db.close();
```

© Editions ENI - All rights reserved

La base de données est donc opérationnelle à partir de là. Néanmoins, il reste à prévoir les différentes fonctionnalités attendues. Pour l'application qui nous concerne, il faudra que des données puissent être enregistrées. Par la suite, il devra être possible de les lire. Et enfin, il sera nécessaire de prévoir leur suppression.

Pour l'enregistrement des informations, une méthode nommée `insert` est intégrée :

```
Future<int> insert(Personne personne) async {
  Database db = await instance.db;
  int id = await db.insert(tablePersonne, personne.toMap());
  return id;
}
```

`Insert` est de type `Future`. Son traitement est évidemment asynchrone. La première étape consiste à appeler le *getter* de _db grâce à `instance`. Sur cette instance de `Database` il devient possible alors d'appeler la méthode `insert`. Elle attend deux paramètres. Le premier doit indiquer la table qui est concernée par l'insertion. Le second apporte les données à insérer. Ici, ces données sont fournies sous la forme d'une `Map`. Cela s'avère très pratique puisque les noms de clés correspondent aux noms des colonnes de la table. La clé primaire étant auto-incrémentée, le retour de la méthode `insert` sera l'`id` qui prend la forme d'un entier.

La seconde fonctionnalité importante doit permettre de lire les données contenues dans la table. Plus précisément, de lire la dernière donnée enregistrée puisque c'est ce que l'application requiert. Voici cette méthode :

```
Future<Personne> getPersonne(int id) async {
  Database db = await instance.db;
  List<Map> maps = await db.query(tablePersonne,
      columns: [colonneId, colonneNom, colonnePrenom, colonneAge],
      where: '$colonneId = ?',
      whereArgs: [id]);
  if (maps.length > 0) {
    return Personne.fromMap(maps.first);
  }
  return null;
}
```

Le début ressemble à `insert`. C'est une méthode asynchrone qui dans un premier temps fait appel au *getter* de `_db`. La différence principale réside dans la récupération d'un paramètre. Il s'agit d'un numéro d'identifiant (`id`) qui va permettre de rechercher parmi l'ensemble des enregistrements contenus dans la table celui qui correspond.

Le retour attendu est une liste d'objets de type `Map`, chacun d'eux correspondant à une `Personne`.

La méthode `query` de `Database` permet d'indiquer plusieurs paramètres :

- Le nom de la table où s'effectue la recherche.
- `columns` qui attend un tableau des colonnes dont on souhaite remonter les informations.
- Puis, `where` offre la possibilité de placer une condition restrictive afin d'éviter de remonter l'ensemble des lignes. Celle-ci indique, par exemple, que le filtrage va avoir lieu sur une colonne précise en lui transmettant une valeur. Ici, la restriction porte sur la colonne `id`. Mais attention ! La valeur n'est pas précisée directement, mais elle est représentée par un point d'interrogation. Elle est ensuite déportée dans le paramètre `whereArgs`. Cette syntaxe permet d'éviter une faille de sécurité en empêchant les injections SQL.

Enfin, une condition permet de vérifier qu'il existe bien au moins une correspondance dans la table. Si c'est le cas, `Personne.fromMap` est utilisée pour construire une `Personne` à partir du premier résultat obtenu.
Sinon, `null` est renvoyé.

La dernière fonctionnalité attendue de la base dans notre exemple est de pouvoir effacer une entrée. Une méthode nommée `delete` est donc créée à cet effet :

```
Future<int> delete(int id) async {
 Database db = await instance.db;
 return await db
     .delete(tablePersonne, where: '$colonneId = ?', whereArgs: [id]);
}
```

© Editions ENI - All rights reserved

Encore une fois, il s'agit d'une méthode asynchrone. La première étape consiste toujours à récupérer une instance de `Database`. Le but est d'appeler sur cette dernière la méthode `delete`. Les paramètres attendus sont les mêmes que pour `query`. D'abord, le nom de la table. Puis une clause `where` avec les arguments de celle-ci dans le paramètre `whereArgs`. L'argument en question est encore une fois l'`id`.

Dans la mesure où cette classe est assez complexe, voici, en récapitulatif, son code complet (accompagné de celui de `Personne` qui se trouve avec) :

```
import 'dart:io';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';

final String tablePersonne = 'personne';
final String colonneId = 'id';
final String colonneNom = 'nom';
final String colonnePrenom = 'prenom';
final String colonneAge = 'age';

final String databaseName = 'PersonneDB.db';
final int databaseVersion = 1;

class Personne {
 int id;
 String nom;
 String prenom;
 String age;

 Personne();

 Personne.fromMap(Map<String, dynamic> map) {
   id = map[colonneId];
   nom = map[colonneNom];
   prenom = map[colonnePrenom];
   age = map[colonneAge];
 }

 Map<String, dynamic> toMap() {
   var map = <String, dynamic>{
     colonneNom: nom,
     colonnePrenom: prenom,
     colonneAge: age
   };
   if (id != null) {
     map[colonneId] = id;
   }
```

```
    return map;
  }
}

class PersonneProvider {

  PersonneProvider._privateConstructor();

  static final PersonneProvider instance =
      PersonneProvider._privateConstructor();

  Database _db;

  Future<Database> get db async {
    if (_db != null) {
      return _db;
    } else {
      _db = await _initDatabase();
      return _db;
    }
  }

  _initDatabase() async {
    Directory documentsDirectory = await getApplicationDocumentsDirectory();
    String path = join(documentsDirectory.path, databaseName);
    print(path);
    return await openDatabase(path, version: databaseVersion, onCreate: _open);
  }

  Future _open(Database db, int version) async {
    await db.execute('''
      create table $tablePersonne (
      $colonneId integer primary key autoincrement,
      $colonneNom text,
      $colonnePrenom text,
      $colonneAge text)
      ''');
  }

  Future<int> insert(Personne personne) async {
    Database db = await instance.db;
    int id = await db.insert(tablePersonne, personne.toMap());
    return id;
  }

  Future<Personne> getPersonne(int id) async {
    Database db = await instance.db;
    List<Map> maps = await db.query(tablePersonne,
        columns: [colonneId, colonneNom, colonnePrenom, colonneAge],
        where: '$colonneId = ?',
        whereArgs: [id]);
```

© Editions ENI - All rights reserved

```
    if (maps.length > 0) {
      return Personne.fromMap(maps.first);
    }
    return null;
  }

  Future<int> delete(int id) async {
    Database db = await instance.db;
    return await db
        .delete(tablePersonne, where: '$colonneId = ?', whereArgs: [id]);
  }

  Future close() async => _db.close();
}
```

3.3.3 main.dart

Il est maintenant temps de s'atteler à l'interface graphique. Comme cela avait été précisé, l'application sera visuellement la même que dans l'exemple d'utilisation des Shared Preferences. `MyHomePage` étend toujours `StatefulWidget` pour s'appuyer sur l'initialisation avec `initState` et la mise à jour de l'interface grâce au `setState`.

Voici la base de code :

```
import 'package:flutter/material.dart';
import 'package:fluttersqflite/personne.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo SQfLite'),
    );
  }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);
```

```
  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
  class _MyHomePageState extends State<MyHomePage> {
...
```

La première chose à faire ensuite est la déclaration des variables qui vont servir durant tout le process.

```
Map<String, dynamic> mapPersonne = {
 'nom': 'DOE',
 'prenom': 'John',
 'age': '35'
};
Map<String, dynamic> mapPersonneRecuperee;
Personne personneEnregistree = Personne();
Personne personneRecuperee = Personne();
PersonneProvider provider;
```

Pour commencer, il faut des données. Les fournir sera le rôle de l'objet `mapPersonne`. Puis comme dans l'exemple précédent, deux instances de `Personne` seront manipulées : `personneEnregistree` et `personneRecuperee`. Pour exposer le retour de la méthode `getPersonne` de la classe `PersonneProvider`, la variable `mapPersonneRecuperee` sera à la tâche. Enfin, une instance de `PersonneProvider` va servir à l'appel des fonctionnalités définies dans cette classe.

Avant de découvrir les différentes actions exécutées, il est préférable de regarder le code de la partie graphique. Celui-ci est très proche de l'application précédente même si quelques changements sont intervenus.

Le début est similaire. Un `Scaffold`, une `AppBar` avec le titre et un `Center/Column` dans le `body`. Le premier widget placé ensuite est une icône :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
```

© Editions ENI - All rights reserved

```
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
        children: <Widget>[
          Icon(
            Icons.account_circle,
            size: 80.0,
            color: Colors.blue,
          ),
```

Vient ensuite le premier cadre dans lequel les données à soumettre apparaissent :

```
Text('Données soumises :'),
Container(
 height: MediaQuery.of(context).size.height * 0.25,
 width: MediaQuery.of(context).size.width * 0.8,
 decoration: BoxDecoration(
     color: Colors.grey,
     shape: BoxShape.rectangle,
     borderRadius: BorderRadius.circular(20.0)),
 child: Column(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
   children: <Widget>[
     Row(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'Nom : ',
           style: TextStyle(color: Colors.white),
         ),
         Text('${personneEnregistree.toMap()['nom']}',
             style: TextStyle(
                 fontSize: 30.0,
                 color: Colors.white,
                 fontWeight: FontWeight.bold)),
       ],
     ),
     Row(
       mainAxisAlignment: MainAxisAlignment.center,
       children: <Widget>[
         Text(
           'Prénom : ',
```

```
            style: TextStyle(color: Colors.white),
          ),
          Text('${personneEnregistree.toMap()['prenom']}',
              style: TextStyle(
                  fontSize: 30.0,
                  color: Colors.white,
                  fontWeight: FontWeight.bold)),
        ],
      ),
      Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Age : ',
            style: TextStyle(color: Colors.white),
          ),
          Text('${personneEnregistree.toMap()['age']}',
              style: TextStyle(
                  fontSize: 30.0,
                  color: Colors.white,
                  fontWeight: FontWeight.bold)),
        ],
      ),
    ],
  ),
),
```

Le petit changement réside ici dans l'appel aux données de l'instance `personneEnregistree`. Elles se font par l'intermédiaire de la méthode `toMap`.

Le second cadre ne présente pas beaucoup de nouveautés non plus.

```
Text('Données récupérées :'),
Container(
 height: MediaQuery.of(context).size.height * 0.25,
 width: MediaQuery.of(context).size.width * 0.8,
 decoration: BoxDecoration(
     color: Colors.blue,
     shape: BoxShape.rectangle,
     borderRadius: BorderRadius.circular(20.0)),
 child: Column(
   mainAxisAlignment: MainAxisAlignment.spaceEvenly,
   children: <Widget>[
```

© Editions ENI - All rights reserved

```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(
      'id : ',
      style: TextStyle(color: Colors.white),
    ),
    personneRecuperee.toMap()['id'] != null
        ? Text('${personneRecuperee.toMap()['id']}',
            style: TextStyle(
                fontSize: 25.0,
                color: Colors.white,
                fontWeight: FontWeight.bold))
        : Text(
            'Aucune donnée',
            style: TextStyle(color: Colors.white),
          )
  ],
),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(
      'Nom : ',
      style: TextStyle(color: Colors.white),
    ),
    personneRecuperee.toMap()['nom'] != null
        ? Text('${personneRecuperee.toMap()['nom']}',
            style: TextStyle(
                fontSize: 25.0,
                color: Colors.white,
                fontWeight: FontWeight.bold))
        : Text(
            'Aucune donnée',
            style: TextStyle(color: Colors.white),
          )
  ],
),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Text(
      'Prénom : ',
      style: TextStyle(color: Colors.white),
```

```
        ),
        personneRecuperee.toMap()['prenom'] != null
            ? Text('${personneRecuperee.toMap()['prenom']}',
                style: TextStyle(
                    fontSize: 25.0,
                    color: Colors.white,
                    fontWeight: FontWeight.bold))
            : Text(
                'Aucune donnée',
                style: TextStyle(color: Colors.white),
              )
      ],
    ),
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          'Age : ',
          style: TextStyle(color: Colors.white),
        ),
        personneRecuperee.toMap()['age'] != null
            ? Text('${personneRecuperee.toMap()['age']}',
                style: TextStyle(
                    fontSize: 25.0,
                    color: Colors.white,
                    fontWeight: FontWeight.bold))
            : Text(
                'Aucune donnée',
                style: TextStyle(color: Colors.white),
              )
      ],
    ),
  ],
),
),
```

Seule une ligne supplémentaire fait son apparition, l'id. Cet élément peut être intéressant pour vérifier l'évolution de l'enregistrement dans la base de données. L'identifiant étant autoincrémenté, il devrait évoluer à chaque fois que des données seront soumises.

© Editions ENI - All rights reserved

Quant à la zone des boutons, elle est totalement identique. Les noms des méthodes appelées restent les mêmes.

```
Divider(),
Row(
 mainAxisAlignment: MainAxisAlignment.spaceEvenly,
 children: <Widget>[
   FlatButton(
       onPressed: enregistrer,
       color: Colors.green,
       textColor: Colors.white,
       child: Text('Enregistrer')),
   FlatButton(
       onPressed: recuperer,
       color: Colors.blue,
       textColor: Colors.white,
       child: Text('Lire les données')),
   FlatButton(
       onPressed: supprimer,
       color: Colors.red,
       textColor: Colors.white,
       child: Icon(
         Icons.delete,
         color: Colors.white,
       ))
 ],
)
```

Trois boutons toujours qui conduisent à trois actions. L'enregistrement des données se fera par l'appel à la méthode `enregistrer`. Le rappel et la lecture de celles-ci en exécutant `recuperer`. Enfin, la suppression des données surviendra quand la méthode `supprimer` sera à l'office.

Avant de voir ces trois fonctionnalités, il convient de parler de l'initialisation. Dans la méthode `initState`, deux actions ont lieu. La première affecte l'objet `Map` déclaré au départ (`mapPersonne`) à la `personneEnregistree` via `fromMap`. L'autre va récupérer une instance de `PersonneProvider`.

```
@override
void initState() {
 super.initState();
 personneEnregistree = Personne.fromMap(mapPersonne);
 getInstance();
}
```

Pour récupérer une instance de `PersonneProvider`, une méthode nommée `getInstance` est créée. Elle est asynchrone et vient simplement appeler `instance` sur le nom de la classe. Celle-ci, `static`, exécutera le contrôleur privé.

```
Future<PersonneProvider> getInstance() async {
 provider = await PersonneProvider.instance;
}
```

Dernière chose. Le cycle de vie de `StatefulWidget` prévoit aussi l'appel de `dispose`. C'est le bon moment pour clore la base de données. Maintenant qu'une instance de `PersonneProvider` est disponible, il est possible d'effectuer cette opération :

```
 @override
void dispose() {
 super.dispose();
 provider.close();
}
```

Il ne reste plus qu'à détailler les trois actions liées aux boutons.

Commençons par l'enregistrement. Voici le code qui va être expliqué juste après :

```
Future<void> enregistrer() async {
 int id = await provider.insert(personneEnregistree);
 mapPersonneRecuperee = {'id': id};
 personneRecuperee = Personne.fromMap(mapPersonneRecuperee);
 showDialog(
   context: context,
   builder: (BuildContext context) {
     return AlertDialog(
       title: Text('Enregistrement'),
       content: Text('Les données ont été enregistrées !'),
     );
   });
}
```

© Editions ENI - All rights reserved

La méthode est asynchrone de fait. La méthode `insert` est appelée sur l'instance `provider` de `PersonneProvider`. Le retour est un entier qui va être affecté à `mapPersonneRecuperee`. Puis `personneRecuperee` est alimentée grâce à `fromMap`. Grâce au champ `id`, il va être possible de lire la ligne en base de données. Pour finir, une `AlertDialog` confirme la réussite de l'enregistrement.

La lecture des données prend cette forme :

```
Future<void> recuperer() async {
 if (personneRecuperee.toMap()['id'] != null) {
   personneRecuperee =
       await provider.getPersonne(personneRecuperee.toMap()['id']);
   setState(() {});
 } else {
   showDialog(
       context: context,
       builder: (BuildContext context) {
         return AlertDialog(
           title: Text('Récupération'),
           content: Text('Aucune donnée à récupérer !'),
         );
       });
 }
}
```

Maintenant que `personneRecuperee` possède l'identifiant, la méthode `getPersonne` peut être invoquée. Cependant, il convient d'éviter cet appel si la valeur de l'`id` n'est pas connue. Ce sera le cas au démarrage de l'application avant qu'un enregistrement ait eu lieu ou après la suppression des données. La tentative de lecture de données inexistantes entraînerait une erreur. La mise en place de la condition protège de cet écueil. Si malgré tout, un appui sur le bouton intervient, une `AlertDialog` s'ouvrira pour prévenir de l'absence de données (figure 6).

Dans le cas où l'id est connu, `getPersonne` le recevra en paramètre et permettra d'affecter `personneRecuperee` en retour. L'exécution de `setState` declenchera à nouveau le `build` et l'affichage sera actualisé (figure 5).

Enfin, la suppression des données correspond à ce code :

```
Future<void> supprimer() async {
  await provider.delete(personneRecuperee.toMap()['id']);
  setState(() {
    personneRecuperee = Personne();
  });
  showDialog(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: Text('Suppression'),
          content: Text('Les données ont été supprimées !'),
        );
      });
}
```

La suppression est effectuée par l'appel de la méthode `delete` sur l'instance `provider`. La valeur de l'`id` est transmise en paramètre. À ce stade, `setState` vient réinitialiser l'instance de `personneRecuperee` et mettre à jour l'affichage. Pour finir, une `AlertDialog` confirme l'effacement des données.

3.4 Rendu de l'application

Le rendu est donc le même que dans l'exemple précédent (Shared Preferences). Toutefois, l'écran présente deux modifications. D'abord, une ligne supplémentaire apparaît dans le cadre du bas. Il s'agit de l'id qui permet de suivre le rythme des enregistrements dans la base de données SQLite.

© Editions ENI - All rights reserved

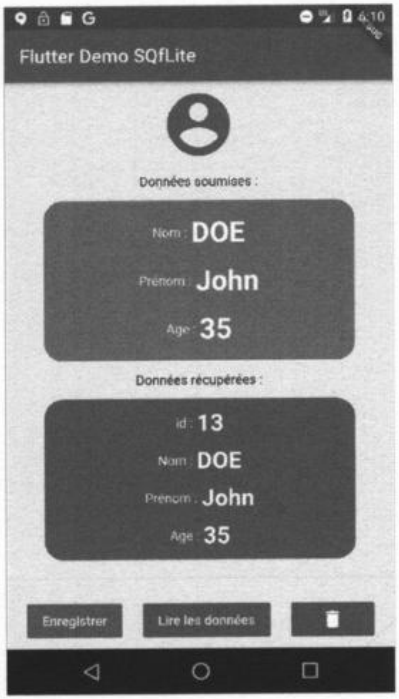


Figure 5 - Écran de récupération des données de la base SQLite

L'autre différence est le popup qui prévient de l'absence de données. Il apparaît si le bouton **Lire les données** est appuyé alors qu'aucun enregistrement n'a eu lieu ou que les données ont été effacées.

Figure 6 - Message d'avertissement d'absence de données

4. JSON

Le format JSON est très couramment utilisé de nos jours pour transmettre de l'information.

4.1 Définition

JSON signifie ***J**ava**S**cript **O**bject **N**otation*.

Il s'agit de données organisées sous un format précis et de manière structurée. Ce format est, à ce titre, un concurrent du XML.

© Editions ENI - All rights reserved

Dans les fichiers JSON, les données sont travaillées autour du principe clé/valeur. Les valeurs peuvent être des objets, des tableaux ou des types plus génériques tels que des nombres, des chaînes de caractères, des booléens…

L'avantage du standard JSON réside dans le fait qu'il est facilement lisible et compréhensible pour un humain puisqu'il contient du texte. Son format permet aux systèmes de l'interpréter également sans problème (des interpréteurs existent pour presque tous les langages). Cette syntaxe (clé/valeur) est plus légère que celle du XML (balise) et les fichiers JSON sont donc moins lourds. Ce qui est important si les données proviennent du réseau.

Voici un exemple de données structurées en JSON :

```
{
    "personne": [

        {
            "id": "1",
            "nom": "DOE",
            "prenom": "John"
        },

        {
            "id": "2",
            "nom": "DOE",
            "prenom": "Jane"
        }

    ]
}
```

Dans l'exemple ci-dessus, des informations contenant deux personnes sont transmises. Le fichier contient un tableau de personnes. Chacune comporte un `id`, un `nom` et un `prenom`.

4.2 Récupération des dépendances

En soi, il n'y a pas de dépendances à récupérer pour manipuler JSON avec Dart. En réalisant l'importation de `dart:convert`, il est possible d'accéder à deux méthodes permettant l'encodage en JSON et son décodage.

```
import 'dart:convert' as convert;
```

Pour rendre les choses un peu plus sympathiques et puisque l'utilisation de JSON est très souvent consécutive à une requête HTTP effectuée auprès d'un serveur, l'application développée en exemple ci-après va suivre ce chemin.

L'idée va être de sonder une base de données de cinéma pour récupérer les informations relatives à un film et les afficher. Les données seront reçues en JSON.

De fait, il faut pouvoir communiquer avec un serveur et donc émettre des requêtes HTTP. Ceci requiert la mise en place d'une dépendance. Le package **http** le permet. Il se situe à cette adresse : https://pub.dev/packages/http

Dans le fichier *pubspec.yaml*, il convient de rajouter ceci :

```
dependencies:
 http: ^0.12.0+4
```

Et comme à chaque fois, il ne faut pas oublier de cliquer sur **Packages get** pour récupérer les éléments.

4.3 Création de l'application

4.3.1 API

Les données vont provenir d'une source ouverte. Elles sont donc exploitables gratuitement. Il existe plusieurs solutions. Celle retenue ici est Open Movie Database. Il s'agit d'un *webservice*, d'une API qui, en fonction des requêtes émises, renverra une réponse contenant une structure JSON.

Le site est disponible à cette URL : http://www.omdbapi.com/

© Editions ENI - All rights reserved

Pour utiliser ce service, il suffit de demander l'attribution d'une clé. En effet, le service est gratuit, mais limité. Pour une utilisation à des fins démonstratives, la version gratuite est plus que suffisante. Le nombre de requêtes possibles est de 1000 par jour. Au-delà, il faudra contribuer financièrement.

Pour que le système soit au courant du niveau de sollicitation, il délivre à chaque utilisateur une clé. Pour l'obtenir, il suffit de cliquer sur le lien **API Key** de la page d'accueil, puis de choisir **FREE !** et de renseigner les quelques champs (figure 7).

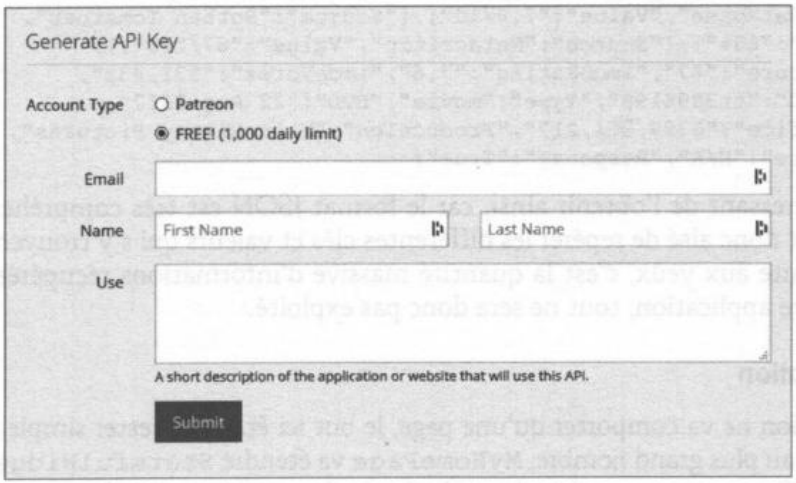


Figure 7 - Inscription sur Open Movie Database

À partir de là, un e-mail contenant la clé sera reçu. Un lien sera mis à disposition également pour activer cette clé. Enfin, un autre lien pointant vers un exemple est aussi présent. En cliquant dessus, un flux au format JSON est affiché dans le navigateur :

```
{"Title":"Guardians of the Galaxy Vol. 2","Year":"2017","Rated":
"PG-13","Released":"05 May 2017","Runtime":"136 min","Genre":
"Action, Adventure, Comedy, Sci-Fi","Director":"James Gunn",
"Writer":"James Gunn, Dan Abnett (based on the Marvel comics by),
Andy Lanning (based on the Marvel comics by), Steve Englehart
(Star-Lord created by), Steve Gan (Star-Lord created by),
Jim Starlin (Gamora and Drax created by), Stan Lee (Groot created
by), Larry Lieber (Groot created by), Jack Kirby (Groot created
```

by), Bill Mantlo (Rocket Raccoon created by), Keith Giffen (Rocket Raccoon created by), Steve Gerber (Howard the Duck created by), Val Mayerik (Howard the Duck created by)","Actors":"Chris Pratt, Zoe Saldana, Dave Bautista, Vin Diesel","Plot":"The Guardians struggle to keep together as a team while dealing with their personal family issues, notably Star-Lord's encounter with his father the ambitious celestial being Ego.","Language":"English", "Country":"USA","Awards":"Nominated for 1 Oscar. Another 14 wins & 52 nominations.","Poster":"https://m.media-amazon.com/images/M/MV5BNjM0NTc0NzItM2FlYS00YzEwLWE0YmUtNTA2ZWIzODc2OTgxXkEyXkFqcGdeQXVyNTgwNzIyNzg@._V1_SX300.jpg","Ratings":[{"Source":"Internet Movie Database","Value":"7.6/10"},{"Source":"Rotten Tomatoes", "Value":"85%"},{"Source":"Metacritic","Value":"67/100"}], "Metascore":"67","imdbRating":"7.6","imdbVotes":"531,435", "imdbID":"tt3896198","Type":"movie","DVD":"22 Aug 2017", "BoxOffice":"$389,804,217","Production":"Walt Disney Pictures", "Website":"N/A","Response":"True"}

C'est intéressant de l'obtenir ainsi, car le format JSON est très compréhensible. Il est donc aisé de repérer les différentes clés et valeurs qui s'y trouvent. Ce qui saute aux yeux, c'est la quantité massive d'informations récupérées. Dans notre application, tout ne sera donc pas exploité.

4.3.2 L'application

L'application ne va comporter qu'une page, le but ici étant de rester simple et accessible au plus grand nombre. `MyHomePage` va étendre `StatefulWidget` pour bénéficier encore une fois du cycle de vie de ce widget. Au chargement, `initState` va lancer la requête HTTP.

Voici le début de l'application :

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert' as convert;

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Demo',
```

© Editions ENI - All rights reserved

```
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo JSON API'),
    );
  }
}

class MyHomePage extends StatefulWidget {
 MyHomePage({Key key, this.title}) : super(key: key);

 final String title;

 @override
 _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
...
```

Deux importations sont ajoutées. La première vers le package `http`, la seconde vers `dart:convert`. Chacune se voit attribuer un alias grâce au mot-clé `as`, respectivement `http` et `convert`, ce qui permettra de l'utiliser plus facilement par la suite.

Dans `_MyHomePageState`, trois variables vont être déclarées.

```
final String url = 'http://www.omdbapi.com/?i=tt3896198&apikey=VOTRE_CLE_ICI';
Map<String, dynamic> film;
bool dataOK = false;
```

La première est l'URL vers laquelle la requête va se faire. Attention de bien remplacer la valeur de `apikey` par la valeur de la clé reçue précédemment par courriel.

La seconde est une variable de type `Map`. Quand le flux JSON sera renvoyé, les données seront décodées et affectées à cette `Map`.

Enfin, un booléen (`dataOK`) va permettre de contrôler que ces données sont bien arrivées. Le processus de récupération peut parfois prendre du temps. Afin d'éviter un affichage « vide » pendant quelques secondes une icône de chargement va être mise en place. Par défaut, sa valeur est donc à `false`.

Avant d'aborder la partie graphique, il faut récupérer les données. Au moment du lancement de l'application, `initState` va exécuter ce code :

```
@override
void initState() {
 recupFilm();
 super.initState();
}
```

Il appelle donc `recupFilm`. Voici son code qui va être détaillé juste après :

```
Future<void> recupFilm() async {
 var response = await http.get(url);
 if (response.statusCode == 200) {
   film = convert.jsonDecode(response.body);
   setState(() {
     dataOK = !dataOK;
   });
 }
}
```

La méthode est asynchrone. En effet, cette étape peut prendre du temps. La première ligne est donc l'envoi de la requête avec par conséquent l'attente d'une réponse. Cette requête est émise grâce à la méthode `get` associée à l'alias `http`. Le paramètre passé est la variable `url` préalablement définie.

Puisque le protocole utilisé est le HTTP, la réponse possèdera un code de statut. Il est donc possible de l'analyser pour vérifier que tout s'est bien passé. Le code 200 indique justement que tout est OK.

La condition ainsi posée permet d'une part de décoder le flux JSON reçu, mais aussi de faire évoluer le booléen `dataOK`.

Les données JSON sont contenues dans le corps de la réponse (`response.body`). Pour le lire et l'affecter à la variable `film`, la méthode `jsonDecode` de `convert` est utilisée. Son paramètre est par conséquent `response.body`, c'est-à-dire le corps de la réponse qui fait suite à notre requête.

Quant au booléen `dataOK`, il évolue au sein de `setState` afin de déclencher un nouveau `build`.

Au sujet de ce dernier, voyons maintenant la partie graphique de l'application.

© Editions ENI - All rights reserved

Voici donc le code du build :

```
@override
Widget build(BuildContext context) {
 return Scaffold(
   appBar: AppBar(
     title: Text(widget.title),
     backgroundColor: Colors.black,
   ),
   body: dataOK ? affichage() : attente(),
   backgroundColor: Colors.blueGrey[900],
 );
}
```

Ce code est très succinct. Il pose une AppBar et un body qui, en fonction de l'état de dataOK, affichera soit affichage ou attente. L'utilisation de l'opérateur ternaire permet d'alterner le widget affiché à l'écran. Puisqu'au début dataOK est à false, c'est attente qui va être chargé.

```
Widget attente() {
 return Center(
   child: Column(
     mainAxisAlignment: MainAxisAlignment.center,
     children: <Widget>[
       Text('En attente des données',
           style: TextStyle(
               color: Colors.white,
               fontSize: 20.0,
               fontWeight: FontWeight.bold)),
       CircularProgressIndicator(),
     ],
   ),
 );
}
```

La méthode attente renvoie un Widget composé d'un Center avec une Column, au sein desquels un texte et un widget CircularProgressIndicator vont venir se placer. Ce dernier est utilisé avec sa configuration par défaut. Il s'agit d'un cercle bleu animé simulant le chargement ou l'attente d'un élément.

Dès que la réponse parvient à l'application et que son code de retour est 200, le booléen `dataOK` passe à `true` dans `setState`. En conséquence, un nouveau `build` est déclenché et cette fois-ci, c'est la méthode `affichage` qui s'exécute.

```
Widget affichage() {
 return Center(
   child: Column(
     mainAxisAlignment: MainAxisAlignment.center,
     children: <Widget>[
       film != null
           ? Text(
               '${film['Title']}',
               style: TextStyle(
                   color: Colors.white,
                   fontSize: 25.0,
                   fontWeight: FontWeight.bold),
             )
           : Text('Aucune donnée'),
       film != null
           ? Text(
               '${film['Year']}',
               style: TextStyle(color: Colors.white),
             )
           : Text('Aucune donnée'),
       Padding(padding: EdgeInsets.all(20.0)),
       Container(
         width: MediaQuery.of(context).size.width * 0.6,
         decoration: BoxDecoration(boxShadow: [
           BoxShadow(
               color: Colors.black,
               offset: Offset(0.0, 7.0),
               spreadRadius: 3.0,
               blurRadius: 15.0)
       ]),
       child: film != null
           ? Image.network('${film['Poster']}')
           : Text('Aucune donnée'),
       ),
       Padding(padding: EdgeInsets.all(20.0)),
       film != null
           ? Text(
               '${film['Plot']}',
```

© Editions ENI - All rights reserved

```
                    style: TextStyle(color: Colors.white),
                    textAlign: TextAlign.center,
                  )
                : Text('Aucune donnée'),
        ],
      ),
    );
  }
```

La méthode `affichage` renvoie aussi un `Widget` qui comprend une association `Center` et `Column`. Parmi le choix important de données renvoyées dans le flux JSON, quatre seront exploitées. En premier lieu, le titre du film se placera en haut de l'écran, suivi juste en dessous de l'année de celui-ci. Au centre, l'affiche du film occupera la majeure partie de l'espace. Pour finir, le résumé du film se situera en bas de l'écran.

Les données ayant été affectées à la variable `Map` nommée `film`, il suffit d'y appeler les clés désirées. En faisant référence au flux JSON écrit un peu avant, nous pouvons constater que le titre possède la clé `Title`, l'année `Year`, l'affiche du film est représentée sous la clé `Poster` et enfin le résumé lui, est associé à la clé `Plot`.

L'affiche du film fait référence à une URL à laquelle se situe l'image. Le constructeur nommé `Image.network` permet de récupérer cette image en ligne.

4.4 Rendu de l'application

Pendant la phase de chargement qui est normalement assez rapide, l'écran de chargement apparaît :

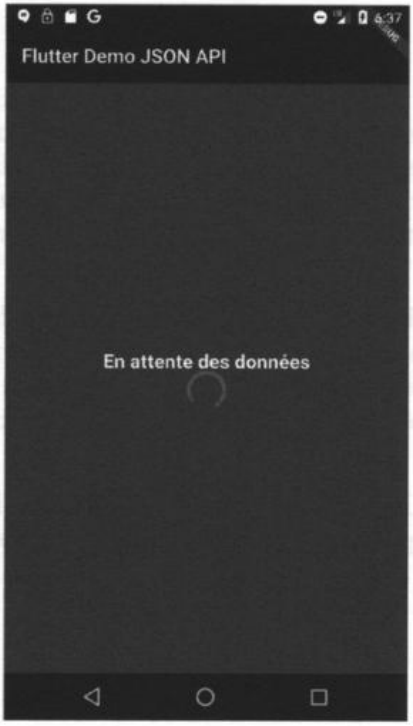


Figure 8 - Écran de chargement des données

© Editions ENI - All rights reserved

Enfin, quand les données parviennent à l'application, la page se met à jour et laisse apparaître les informations relatives au film *'Les Gardiens de la Galaxie 2'*.

Figure 9 - Écran avec les données exploitées du JSON

!

@override, 139

A

abstract, 138
Accéléromètre, 410
AlertDialog, 380, 420, 443
Alignment, 245
Android Studio, 60
    run, 83
Android Virtual Device Manager
    Voir AVD Manager
AndroidManifest.xml, 67, 398, 406
AppBar, 73, 319
AppDelegate.swift, 68
Assets, 68
async, 159
Attributs privés, 121
AVD Manager, 75
await, 159, 340

B

Bannière de Debug, 91
barrierDismissible, 285
bash, 35
Batterie, 410
body, 73
bool, 102

BottomNavigationBarItem, 172, 180
BottomSheet, 347
BoxDecoration, 239, 242, 326, 330
boxShadow, 326
build, 72, 256, 263
BuildContext, 285, 291, 372, 381
Builder, 282

C

Card, 393
catch, 152
ChangeNotifierProvider, 269
CircularProgressIndicator, 453
clean, 263
Color, 89, 223
Company domain, 64
const, 104
Container, 170, 324, 334, 369, 376, 388, 415
context, 280
controlsBuilder, 253
createState, 70, 258, 263
CupertinoActionSheet, 306
CupertinoActionSheetActions, 308
CupertinoDatePicker, 347
CupertinoDialogAction, 289
CupertinoIcons, 209
CupertinoPageScaffold, 179
CupertinoSlider, 368
CupertinoSwitch, 358

© Editions ENI - All rights reserved

CupertinoTabBar, 180
CupertinoTabScaffold, 180
CupertinoTabView, 182
CustomChildLayout, 256
CustomPaintWidget, 256

D

dart
    convert, 448
    io, 400
DartPad, 98
DateTime, 340
debugShowCheckedModeBanner, 91
decoration, 239
DefferredLoadException, 151
didChangeDependencies, 263
didUpdateWidget, 263
dirty, 263
Dismissible, 381
dispose, 263, 277, 336, 442
Divider, 303, 360, 381, 394, 418
double, 99
DrawerHeader, 176
dynamic, 102

E

EdgeInsets, 247
EdgeInsetsDirectionnal, 247
EgdeInsetsGeometry, 247
elevation, 286
Émulateur, 56
    iPhone, 42
Émulateur Android, 74
    paramètres, 82
    performances, 81
Encapsulation, 129
Énumération, 295, 360
extends, 133

F

File, 400
final, 104
finally, 155
FlatButton, 287, 289
flex, 237
floatingActionButton, 271
    child, 73
    onPressed, 73
    tooltip, 73
floatingActionButtonLocation, 166
Flutter Application, 62
flutter doctor, 75
Flutter Favorite, 265, 412
flutter upgrade, 50
flutter --version, 48

© Editions ENI - All rights reserved

Fonction fléchée, 69
fontFamily, 227
fontSize, 223
fontStyle, 223
fontWeight, 223
forEach(), 118
FormatException, 151, 156
Future, 159, 294, 337, 345, 420, 431

G

Geolocator, 405
GlobalKeys, 257
gradient, 388
Gradle, 67
gridDelegate, 391
groupValue, 359
Gyroscope, 410

H

Highlight, 386
hintText, 322
home, 70
hot reload, 87, 309
HTTP, 448

Icon, 209
IconButton, 170
Icons, 209
iconSize, 200
IDE, 50
Image.asset, 220
Image.file, 403
Image.network, 221, 455
image_picker, 398
ImagePicker.pickImage, 402
implements, 139
import, 121, 131
IndexedWidgetBuilder, 372
Info.plist, 68, 398, 406
InheritedWidget, 256
initState, 263, 336, 406, 421, 435, 441, 450, 452
Ink, 201
InputDecoration, 322
int, 99
IntegerDivisionByZeroException, 151
Interpolation, 117
IOException, 151
isDismissible, 303
IsolateSpawnException, 151

© Editions ENI - All rights reserved

K

keyboardType, 333

L

leading, 183
Licence Android, 56
LinearGradient, 388
List, 105
localizations, 341
localizationsDelegates, 342
Location, 405

M

main, 69, 114
MainAxisAlignment, 231
mainAxisAlignment, 236
Map, 314, 363, 428, 432, 451
MaterialPageRoute, 311
MediaQuery.of(context), 324
middle, 183
mounted, 263
MultiProviders, 271

N

Navigator, 310
Navigator.of(context).pop(), 288
Navigator.pop, 295
new, 126
notifyListeners, 267
num, 100

O

onChanged, 325, 336, 364
onPressed, 87, 166, 193, 195, 197, 200, 204, 288, 345, 418
onSubmitted, 325
onTap, 385

P

Package name, 65
Padding, 211, 243, 313, 326, 331, 341, 345
Path, 426
Path_provider, 426
Permissions, 398
placeholder, 330
Platform, 187
Plug-ins, 56
print(), 101, 115
private, 121
Projet Flutter, 60
Provider.of<T>(context), 270

© Editions ENI - All rights reserved

pubspec.yaml, 68, 214, 219, 229, 266, 341, 374, 398, 412, 426, 448

R

RaisedButton, 282, 313, 338
Random, 260
RenderFlex overflowed, 323
Réseau mobile, 410
Ressources binaires, 34
return, 115, 119
routes, 314
Row, 216
runApp, 69
runtimeType, 103, 150

S

Scaffold, 72, 183, 280
Scaffold.of(context), 282
ScaffoldState.showBottomSheet, 301
separatorBuilder, 381
setState, 72, 87, 164, 182, 258, 261, 263, 264, 295, 355, 360, 376, 402, 452
shape, 197
    CircularNotchedRectangle, 171
ShapeDecoration, 239, 243
showCupertinoModalPopup, 308
showDatePicker, 337
showDialog, 291, 295
showModalBottomSheet, 301, 303, 349

showSnakBar, 280
showTimePicker, 344
SimpleDialogAction, 291
SimpleDialogOption, 295
SingleChildScrollView, 323, 372
sink, 272
snackBar, 376
SnackBarAction, 280
Snapshot, 276
Splash, 386
sqflite, 426
StatefulWidget, 70, 250, 274, 291, 322, 338, 373, 399, 406, 413, 435, 450
StatelessWidget, 70, 283, 392
static, 429, 442
Step, 252
Stream, 159, 272
StreamBuilder, 275
StreamController, 272
String, 100
style, 223
substring(), 102
supportedLocales, 342
switch, 295, 303
Switch/Case, 175, 181, 308

© Editions ENI - All rights reserved

T

tabBuilder, 180
Terminal, 31, 33
Test, 68
textAlign, 223
TextInputType, 333
TextOverflow, 223, 226
TextStyle, 88, 89, 223, 227, 327, 361
throw, 153
TimeOfDay, 345
TimeOfDay.fromDateTime, 353
TimeOutException, 151
title, 70
toLowerCase(), 101
toString(), 100
toUpperCase(), 101
trailing, 183
Transtypage, 135
try, 152
try/catch, 402
tryParse, 100

V

var, 102
Version d'Android
    Android 10, 78
    Lollipop, 78
    Marshmallow, 78
    Nougat, 78

Oreo, 78
Pie, 78
video_player, 404
void, 114

W

Widget, 403
WidgetBuilder, 314
Wi-Fi, 410
with, 141

X

Xcode, 57, 74

Z

zsh, 35, 37

© Editions ENI - All rights reserved

INFORMATIQUE
TECHNIQUE

Flutter™

Développez vos applications mobiles multiplateformes avec Dart

Ce livre sur **Flutter** s'adresse aux **développeurs**, initiés comme plus aguerris, qui souhaitent disposer des connaissances nécessaires pour **créer de A à Z des applications mobiles multiplateformes** avec le framework de Google. Tout au long du livre, l'auteur enrichit ses propos à l'aide d'un grand nombre de **démonstrations simples à reproduire**, permettant au lecteur d'assimiler les connaissances par la pratique.

Après une brève introduction sur le **monde du développement mobile** et une **présentation du framework**, un chapitre détaille les **étapes d'installation** indispensables pour utiliser le framework dans les meilleures conditions. La **création d'un premier projet** permet d'en étudier plus en détail la structure et le code basé sur le **langage Dart**. Pour être parfaitement à l'aise sur ce langage, deux chapitres viennent enrichir les connaissances du lecteur avec des notions fondamentales puis plus avancées.

Avec Flutter, tout est widget ! Ainsi, la suite du livre détaille cette notion fondamentale de widget et en présente les principaux. Puis l'auteur étudie la **gestion des états** qui permettent notamment la mise à jour en temps réel de l'affichage. Les interactions avec l'utilisateur étant indispensables, les **pop-ups**, la **navigation** et les **formulaires** font également l'objet de chapitres dédiés. Un chapitre sur les **listes** et les **grilles** permet au lecteur de voir comment optimiser l'affichage de son application.

L'accès aux outils de l'appareil étant primordial pour une application mobile, l'auteur explique comment exploiter les **API** pour utiliser l'appareil photo et le GPS. Pour finir, il détaille la **persistance des données** sous différentes formes.

Des éléments complémentaires sont en téléchargement sur le site www.editions-eni.fr.

Julien TRILLARD

Formateur en développement web et dirigeant d'une entreprise spécialisée dans le développement logiciel et la formation, **Julien TRILLARD** fait de la création de logiciels et d'applications web et mobiles son quotidien. Au travers de ses expériences de conception d'applications mobiles, il découvre le framework Flutter auquel il voue aujourd'hui une véritable passion. Dans ce livre, il transmet au lecteur tout son retour d'expérience sur le développement d'applications multiplateformes avec ce framework de Google.

Sur **www.editions-eni.fr** :
➜ Le code des exemples et des démonstrations du livre

Pour plus d'informations :

39 €

ISBN : 978-2-409-02527-3
9 782409 025273

eni
www.editions-eni.fr