

« Mobile & Image »

« Développement d'une application mobile »

Par Alexis GUILLOTEAU & Didier JUGE-HUBERT

Sommaire

1	Introduction.....	5
1.1	Comment est réalisée une application mobile ?.....	5
1.1.1	Quels sont les avantages d'une application native ?	6
1.1.2	Quels sont les inconvénients d'une application native ?	6
1.2	Mais quel palliatif aux inconvénients d'une application native ?	7
1.2.1	Quels sont les avantages d'une application web mobile ?	7
1.2.2	Quels sont les inconvénients d'une application web mobile ?	7
1.3	Que faire alors : application hybride un bon compromis ?.....	7
1.3.1	Quels sont les avantages d'une application hybride ?	7
1.3.2	Quels sont les inconvénients d'une application hybride ?	8
1.4	Une autre approche, la PWA ?	8
1.4.1	Quels sont les avantages d'une application web avec la PWA ?.....	8
1.4.2	Quels sont les inconvénients d'une application web avec la PWA ?	8
1.5	Est demain que vais-je faire...	8
1.6	Dart et Flutter.....	9
2	L'environnement Dart	10
2.1	Introduction.....	10
2.2	Les évolutions de Dart.....	10
2.3	Comment fonctionne Dart	11
2.3.1	La compilation JIT	12
2.3.2	La compilation AOT	12
2.4	Les IDE de DART.....	12
2.4.1	Dart Editor	12
2.4.2	Dart Pad	13
2.4.3	Plugins	15
2.5	Mon premier programme Dart : Bonjour le monde !	15
2.6	Un monde de productivité	16
2.7	Dart un apprentissage facile.....	17
2.8	La maturité de Dart	18
3	Les bases du langage Dart	20
3.1	Introduction.....	20
3.2	Les opérateurs Dart.....	20
3.2.1	Les opérateurs arithmétiques	20
3.2.2	Les opérateurs d'incrémentation et de décrémentation	21
3.2.3	Les opérateurs d'égalité et relationnels	21
3.2.4	Les opérateurs de vérification de type et conversion	21
3.2.5	Les opérateurs logiques.....	22
3.2.6	Les opérateurs de manipulation de bits.....	22
3.2.7	Les opérateurs « Null-safe » et « Null-aware »	22
3.3	Les types de variables de Dart.....	22
3.3.1	Les méthodes <code>final</code> et <code>const</code>	22

3.3.2	Les types intégrés	23
3.3.3	L'inférence de type	25
3.4	Les instructions et les fonctions	26
3.4.1	Les traitements conditionnels et les boucles	26
3.4.2	Les fonctions.....	27
3.4.3	Les fonctions anonymes	29
3.4.4	La portée des éléments	29
3.5	Structures de données, collections et génériques	29
3.5.1	Comment définir les génériques	30
3.5.2	Quand et pourquoi utiliser des génériques.....	30
3.5.3	Génériques et valeurs littérales	30
3.6	Introduction à la POO dans Dart	31
3.6.1	Fonctionnalités de la POO de Dart	31
 4	 Les aspects avancés du langage Dart	34
4.1	Introduction.....	34
4.2	Les classes Dart.....	34
4.3	Le type enum	35
4.4	La notation en cascade	36
4.5	Les Constructeurs	36
4.5.1	Définition d'un constructeur	36
4.5.2	Les constructeurs nommés.....	37
4.5.3	Le constructeur factory	37
4.5.4	Accesseurs de champs – get et set	38
4.5.5	Les champs et les méthodes statiques.....	40
4.5.6	Héritage de classe.....	41
4.5.7	Interfaces, classes abstraites et mixins	43
4.5.8	Les classes appelables, les fonctions et variables de niveau supérieur	46
4.6	Comprendre les bibliothèques et les packages Dart.....	48
4.6.1	Importer et utiliser une bibliothèque.....	48
4.6.2	Renommer ou masquer des éléments de la bibliothèque	49
4.6.3	Importer des préfixes dans des bibliothèques	49
4.6.4	Définition des répertoires d'importation	51
4.6.5	Création de bibliothèques Dart	52
4.6.6	Les packages Dart	59
4.7	La programmation asynchrone avec <i>Future</i> et <i>Isolate</i>	66
4.7.1	Introduction.....	66
4.7.2	<i>Future</i>	66
4.7.3	<i>Isolates</i>	69
4.8	Présentation des tests unitaires avec Dart.....	70
4.8.1	Introduction.....	70
4.8.2	Le package de test Dart	71
 5	 Flutter	74
5.1	Introduction à Flutter	74
5.2	Comparaisons avec d'autres frameworks de développement d'applications mobiles.....	74
5.2.1	introduction.....	74
5.2.2	Les problèmes que Flutter veut résoudre	74
5.2.3	Différences entre les frameworks existants	75
5.2.4	Compilation Flutter (Dart)	80

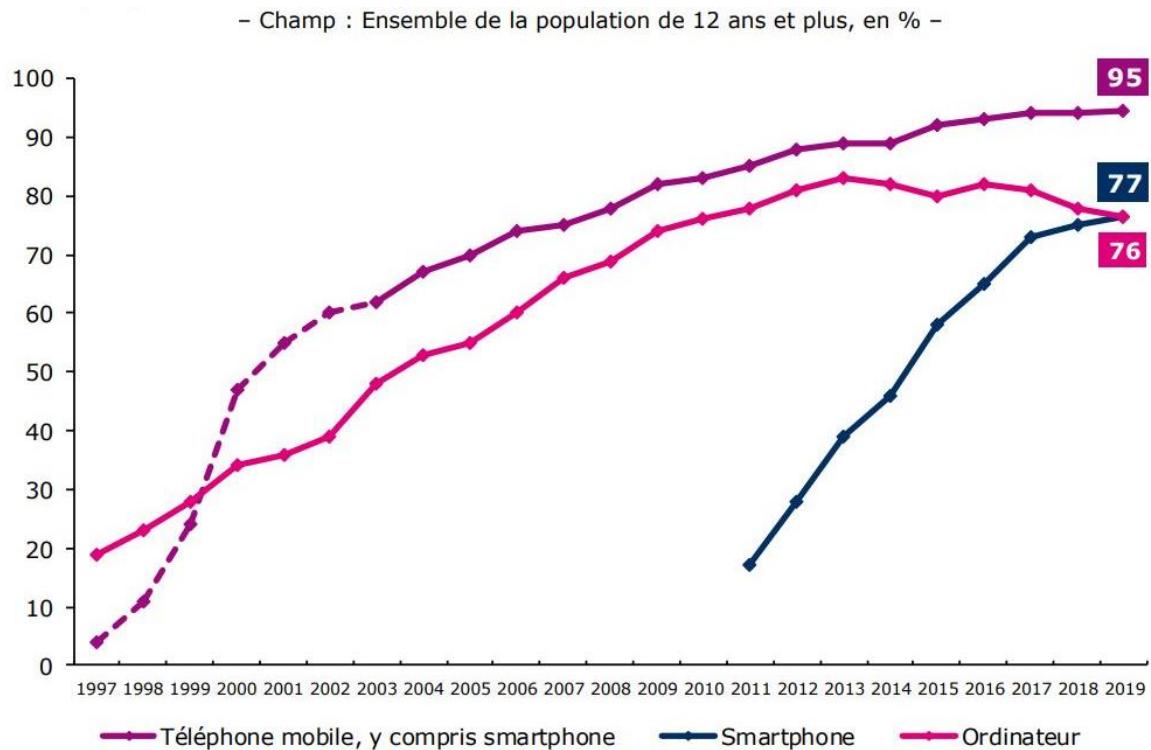
5.2.5	Rendu Flutter.....	80
5.2.6	Introduction aux widgets.....	82
5.2.7	Mon premier programme flutter	84
6	Interface utilisateur de flutter - Tout est widget.....	91
6.1	Widgets : Création d'écrans dans flutter.....	91
6.1.1	Les widgets sans état <i>stateless</i> et avec état <i>stateful</i>	91
6.1.2	Les widgets intégrés	96
6.1.3	L'utilisation des widgets intégrés	100
6.1.4	La création de widgets personnalisés.....	101
6.1.5	Ma deuxième application flutter.....	101
6.2	Gestion des entrées et des gestes de l'utilisateur.....	118
6.2.1	Gestion des gestes de l'utilisateur.....	118
6.2.2	Les widgets d'entrée.....	126
6.2.3	Création d'entrées personnalisées.....	131
6.2.4	Ma deuxième application flutter (suite).....	134
6.3	Thème et style	142
6.3.1	Les widgets de définition d'un thème	142
6.3.2	<i>Material Design</i>	147
6.3.3	iOS Cupertino.....	152
6.3.4	Utilisation de polices personnalisées	154
6.3.5	Style dynamique avec MediaQuery et LayoutBuilder	155
6.4	Navigation entre les écrans.....	162
6.4.1	Comprendre le widget <i>Navigator</i>	162
6.4.2	Comment faire pour tout rassembler	163
6.4.3	Les transitions d'écran.....	173
6.4.4	Les animations Hero	175
7	Accéder à une base de données depuis l'application Flutter.....	182
7.1	Introduction.....	182
7.2	Accéder à la zone de stockage d'une application (<i>SharedPreferences</i>)	182
7.2.1	Implémentation de <i>SharedPreferences</i>	182
7.2.2	Utilisation de <i>SharedPreferences</i>	182
7.3	Accéder à une base de données interne SQLite.....	184
7.3.1	Rappel sur la théorie des Bases de Données Relationnelles (BDR).....	184
7.3.2	Créer une base de données SQLite	185
7.3.3	Création des classes de modèle pour accéder à une base de données	190
7.3.4	Affichage des données de la base de données à l'utilisateur.....	193
8	Accéder au WEB depuis une application Flutter	200
8.1	Introduction.....	200
8.2	Rappel http	200
8.2.1	Outils	201
8.2.2	L'identification du serveur : URI (Uniform Resource Identifier).....	201
8.2.3	Les trames HTTP	201
8.2.4	Les API	202
8.2.5	REST (Representational State Transfer).	202
8.2.6	Le protocole JSON (JavaScript Object Notation)	204
8.3	Flutter avec HTTP, API, REST et JSON	204

8.3.1	Flutter & JSON	204
8.3.2	Sérialisation et désérialisation de JSON	205

1 Introduction

Comme des milliards d'individus dans le monde, vous utilisez votre téléphone quotidiennement pour faire plein de choses : vous connecter aux réseaux sociaux, acheter sur les sites marchands, accéder à vos comptes bancaires, payer vos achats ...

Mais revenons à notre petit pays, en France toutes ces actions sont souvent réalisées par l'utilisation, sur votre smartphone ou tablette, d'une « application mobile ».



Source : CREDOC, Enquêtes sur les « Conditions de vie et Aspirations ».

1.1 Comment est réalisée une application mobile ?

Nous allons dans un premier temps faire le point sur les technologies majeures de développement d'applications mobiles. Dans un premier temps, nous disposons de différents langages informatiques qui ont évolué au fil du temps pour prendre en compte et s'adapter aux fonctionnalités évolutives du mobile.

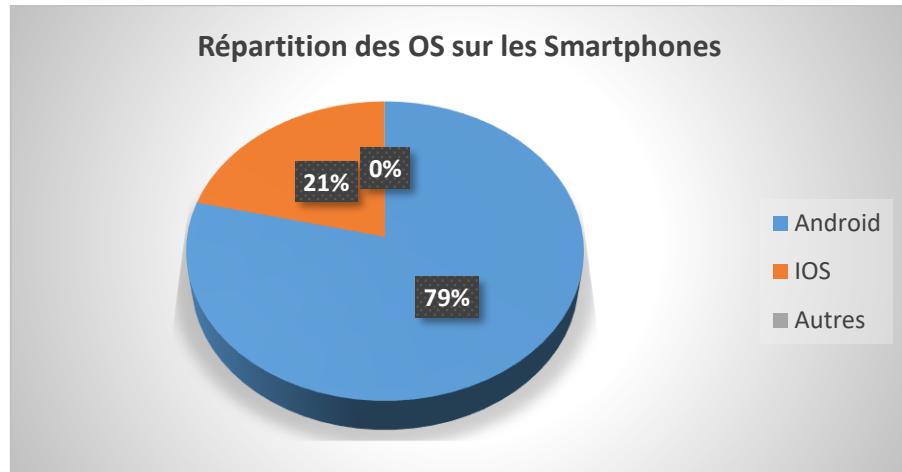
« Au commencement, il y eut le natif ». C'est avec l'application native que tout a commencé. Les applications natives sont développées spécifiquement pour un système d'exploitation (OS : Operating System). Mais quelles sont ces OS ? Sur l'ensemble des smartphones, les OS les plus rencontrés sont les suivants :

- iOS chez Apple,
- Android (Google) chez Samsung, HTC, Huawei, Sony, Xiaomi, Wiko, Honor,
- Windows Phone de Microsoft,
- BlackBerry OS.



Chacun de ces OS utilise ce que l'on appelle un kit de développement logiciel (SDK : Software Development Kit) qui lui est propre.

Pour créer une application mobile, il nous faut alors utiliser le langage de programmation compatible avec l'OS souhaité.



Comme le montre le graphique ci-dessus, deux OS sont les plus usités : IOS et Android.

OS	qui utilisent	anciennement
	 Swift	 Objective-C
	 Kotlin	 Java

1.1.1 Quels sont les avantages d'une application native ?

Le principal avantage de l'application native, c'est son temps de réaction. Du fait que l'appli soit installée directement dans la mémoire du mobile, elle est stable, responsive et exploite parfaitement et rapidement tous ses modules, parfois même sans connexion à internet.

En optimisant le code, elle nous permet de créer une application performante et sur-mesure et de développer des fonctionnalités avancées adaptées aux besoins des utilisateurs. Elle convient donc parfaitement aux projets les plus complexes.

1.1.2 Quels sont les inconvénients d'une application native ?

Hélas, comme dans beaucoup de cas, il y a aussi des inconvénients qui sont d'ordres économiques et techniques :

- Les développeurs capables de créer une application en natif sont rares donc chers.
- Les codes étant radicalement différents d'un OS à un autre, il faut multiplier par deux les coûts de développement pour être présent sur IOS et sur Android.

- Les développeurs sont contraints de s'adapter au code imposé par Apple et Google qui évolue en permanence.
- Les applications natives dépendent des « stores » (boutiques de distribution des applications : AppStore / Google Play) ce qui complique les évolutions et les mises à jour.

Aussi, ces coûts importants et ces fortes contraintes nécessaires pour créer une application native ont donné naissance à de nouvelles technologies.

1.2 Mais quel palliatif aux inconvénients d'une application native ?

La première approche a été d'utiliser sur le mobile une application commune sur tous les OS, par exemple un navigateur internet (browser), puis de développer une application commune sur un serveur, par exemple un site web. L'application web mobile est née.

Généralement, le site web reprend les codes des applications natives et utilise des technologies comme le HTML5 ou le CSS3. Elle est donc compatible avec tous les navigateurs.

1.2.1 Quels sont les avantages d'une application web mobile ?

L'application web mobile ne nécessite aucune installation sur le mobile, donc pas de « pollution » dans la mémoire du smartphone.

En outre, les applications « web mobile » nécessitent moins de développement et leurs déploiements sont rapides et faciles. De plus, les développeurs de ce type d'application sont nombreux sur le marché.

Elles apportent aussi une ouverture sur les évolutions des techniques et du matériel et ne présentent aucun problème de compatibilité avec les systèmes d'exploitation du mobile.

1.2.2 Quels sont les inconvénients d'une application web mobile ?

L'inconvénient majeur de ce type d'application est la nécessité d'avoir une connexion internet pour leur fonctionnement. C'est connexion, dans certain cas, peut entraîner un surcoût non négligeable pour l'utilisateur.

De plus, l'utilisation des fonctionnalités du téléphone reste très limitée, offrant ainsi une expérience utilisateur plus sommaire, moins intuitive et moins interactive. Enfin, de par son aspect généraliste, elle n'est pas non plus optimisée pour tous les formats d'écran.

Créer une application web mobile convient idéalement aux projets peu complexes ne nécessitant pas de faire appel aux nombreuses fonctionnalités internes du mobile (GPS, caméra, carnet d'adresses...).

1.3 Que faire alors : application hybride un bon compromis ?

L'application hybride fonctionne comme une application web mobile à la différence près qu'elle est encapsulée dans une application native, donc présente sur l'écran du téléphone comme n'importe quelle application.

Des passerelles (bridges) raccordent facilement l'application hybride à une grande majorité de fonctions du téléphone. Pour cette raison, nous pouvons dire qu'elle se trouve à mi-chemin entre l'application natice et l'application web mobile.

1.3.1 Quels sont les avantages d'une application hybride ?

Les avantages sont principalement le cumul des avantages des deux premiers types d'applications sous réserve de choisir le bon environnement de développement.

- Facilité et rapidité de développement,
- Coûts moins élevés que pour créer une application native en jouant sur la mutualisation,

- Evolutivité de l'application facilitée puisque l'application est téléchargée sur le support mobile...

1.3.2 Quels sont les inconvénients d'une application hybride ?

Cependant, tous n'est pas rose, il reste quelques inconvénients :

- Les performances seront inférieures à celles d'une application native moins de réactivité,
- Un besoin de maintenance est à prévoir,
- Le code est moins optimisé que dans le natif ce qui ne permet pas de toucher à une utilisation plus poussée des fonctionnalités du smartphone.

1.4 Une autre approche, la PWA ?

La PWA (Progressive Web App) est aussi une alternative aux applications native et web mais elle diffère légèrement d'une application hybride. La PWA permet, en installant un programme léger sur le mobile, de gérer des données en cache, donc de fonctionner offline. Cette technologie web fonctionne ainsi sans encapsulage.

Une application en PWA est, selon Maximiliano Firtman : « *PWA est un modèle de conception pour développer des expériences d'application utilisant des technologies Web avec différentes API, telles que Service Workers pour la gestion des actifs, Web App Manifest pour l'installation du navigateur, Trusted Web Activities pour la distribution sur Play Store* ».

1.4.1 Quels sont les avantages d'une application web avec la PWA ?

Créer une application web mobile avec la technologie PWA apporte un net avantage : elle supprime toute dépendance aux magasins en ligne en permettant de télécharger l'application directement sur internet sans pour autant mobiliser l'espace de stockage du téléphone.

Bien sûr, le client étant dédié, sa rapidité d'exécution est supérieure à celle d'une application web mobile classique et elle s'adapte à tous les navigateurs et tous les systèmes d'exploitation.

1.4.2 Quels sont les inconvénients d'une application web avec la PWA ?

L'inconvénient majeur est qu'il est nécessaire que le système d'exploitation autorise ce type d'application en téléchargement direct.

Google a compris l'importance que peut prendre la PWA et a su rapidement s'adapter en permettant à ces applications d'être disponibles au téléchargement sur leur store Google Play.

Par contre, Apple traîne la patte et ni l'AppStore, ni l'iOS n'ont ouvert leur porte à cette technologie. A suivre ...

En terme de performance, la PWA est plus aboutie que l'hybride et offre une différence majeure : l'utilisation off-line. La mise en cache permet aussi de réduire considérablement le temps de chargement.

1.5 Est demain que vais-je faire...

Il nous manquait un des acteurs majeurs des mobiles : Facebook. Ce dernier veut créer un langage applicatif en s'affranchissant de l'OS et du téléphone support. Pour cela, il a développé de nouvelles technologies dites « natives ».

Dans cette catégorie, nous trouvons également :

- ReactNative de Facebook,
- Flutter de Google,
- NativeScript d'Apple...



A la vue de l'importance que prennent les applications mobiles aujourd'hui dans la sphère du digitale, la mutualisation des ressources de développement prend une part prépondérante.

Les technologies lancées par Facebook permettent de générer du code natif. Nous générerons un seul code source qui sera précompilé n fois pour obtenir n codes natifs. Chacun d'eux pourra être compilé comme une application native par les chaînes de compilation standard. Nous obtenons donc des applis parfaitement natives.

La seule limite : chaque projet est différent et aucune technologie ne peut accomplir automatiquement et de manière efficace 100% du code utile. En fonction des objectifs fonctionnels à atteindre, il est nécessaire de « terminer le code à la main » même si déjà 80 % à 90 % du code sera disponible. En conclusion, un gain de temps et un gain de maintenabilité non négligeable.

Nous allons voir, découvrir, connaître et apprendre l'une de ces dernières technologies développées par Google : Dart et Flutter.

1.6 Dart et Flutter

Le langage Dart est présent au cœur du framework Flutter. Un nouvel outil tel que Flutter nécessite un langage moderne de haut niveau pour être en mesure de fournir la meilleure expérience au développeur et de créer de superbes applications mobiles ou site web.

Comprendre Dart est fondamental pour travailler avec Flutter. Les développeurs doivent connaître les origines du langage Dart, comment la communauté y travaille, ses forces et pourquoi c'est le langage de programmation choisi pour développer avec Flutter.

Dans le premier chapitre, nous verrons les bases du langage Dart. Nous regarderons ensuite une revue des types et opérateurs intégrés à Dart et la façon dont Dart fonctionne avec la programmation orientée objet (POO).

Le but de ce chapitre est que vous compreniez ce que propose le langage Dart. De ce fait, vous pourrez expérimenter confortablement l'environnement Dart et approfondir vos connaissances.

2 L'environnement Dart



2.1 Introduction

Le langage Dart (anciennement appelé Dash) a été développé par Google. Au départ, Dart est un langage de programmation multi applications WEB, de bureau, coté serveur et mobiles.

Dart a été ensuite utilisé pour coder les applications utilisant le framework Flutter. Cette association permet de fournir la meilleure expérience au développeur pour la création d'applications mobiles de haut niveau.

Nous allons dans un premier temps, ce que Dart fournit et comment cela fonctionne afin que nous puissions appliquer plus tard ce que nous apprenons dans Flutter.

Comme beaucoup de langage récent, Dart essaye de garder les avantages de langages plus anciens et d'intégrer des nouvelles fonctionnalités de langage « mature ». Voici quelques exemples :

Outils avancés de développement: analyse du code, plugins d'environnement dans les environnements de développement intégré (IDE : Integrated Development Environment) les plus usités.

Récupérateur de mémoire (Garbage Collector): gère ou traite la désallocation de la mémoire (principalement la mémoire occupée par des objets qui ne sont plus utilisés).

Annotations de type (optionnel) : consiste à vérifier la sécurité et la cohérence des données de l'application.

Type prédéfini : bien que les annotations de type soient facultatives, Dart est de type sécurisé et utilise l'inférence de type pour analyser les types au moment de l'exécution. Cette fonctionnalité est importante pour trouver des bogues pendant la compilation.

Portabilité: ce n'est pas seulement pour le Web (précompilé en JavaScript), mais il peut être compilé nativement en code ARM et x86.

Nous aborderons les sujets suivants dans ce chapitre:

- Connaître les principes et les outils du langage Dart
- Comprendre pourquoi Flutter utilise Dart
- Apprendre les bases de la structure du langage Dart
- Présentation de la POO avec Dart

2.2 Les évolutions de Dart

Né en 2011, Dart évolue depuis. La première version stable de Dart est sortie en 2013. Une première évolution majeure de cette dernière version a eu lieu avec Dart 2.0 vers la fin de 2018.

Dans cette évolution, Dart 2.0 était axé sur le développement Web dans sa conception, avec pour objectif principal de remplacer JavaScript.

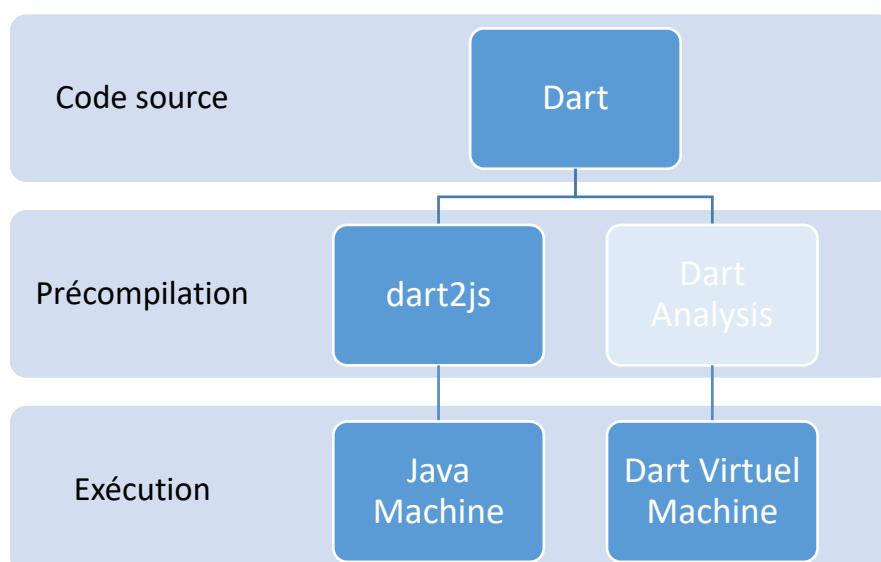
Au fil des années, comme indiqué dans l'introduction sur la poussée des besoins des smartphones, Dart a changé de cible pour se concentrer sur le mobile, avec la création de la première version de framework pour mobile : Flutter.

Dans ces évolutions, Dart a essayé de résoudre les problèmes de JavaScript qui ne fournit pas la robustesse de nombreux langages consolidés. Ainsi, Dart a été lancé en tant que successeur mature de JavaScript. Il offre des meilleures performances et de meilleurs outils pour les projets à grande échelle. En effet, il dispose d'outils modernes et stables fournis par les plugins IDE. Il a été conçu pour obtenir les meilleures performances possibles tout en conservant la sensation d'un langage dynamique. Il a été construit pour être robuste et flexible tout en gardant les annotations de type facultatives et en ajoutant des fonctionnalités POO.

En conclusion, Dart équilibre les deux mondes de flexibilité et de robustesse. Il est un excellent langage moderne multiplateforme et polyvalent qui améliore continuellement ses fonctionnalités, le rendant plus mature et plus flexible.

2.3 Comment fonctionne Dart

Pour comprendre d'où vient la flexibilité du langage, nous devons savoir comment exécuter du code Dart. Le code Dart peut être exécuté suivant deux schémas :



Mais cela ne suffit pas pour qu'un langage soit efficient. Il est nécessaire d'avoir à disposition un écosystème performant. Dart a cette écosystème décrit ci-dessous :

- Des IDE
 - Dart Editor (Dart 1.0)
 - Dart Pad (Dart 1.0 et 2.0)
 - Plugins pour Eclipse, IntelliJ, Visual Studio, ...
- Un SDK avec un VM serveur
- Une VM client : Dartium pour Dart 1.0 / Chrome pour Dart 2.0
- un outil de gestion des dépendances (Pub Package Manager)
- un compilateur Dart vers JavaScript (dart2js)
- un générateur de documentation à partir du code (DartDoc)

Le code Dart peut être exécuté directement dans un environnement compatible Dart qui fournira des fonctionnalités essentielles à une application, telles que les suivantes: systèmes d'exécution, des

bibliothèques et un « Garbage Collector ». Dans ce cas, l'exécution du code Dart fonctionne selon deux modes: la compilation « Just-In-Time (JIT) » ou la compilation « Ahead-Of-Time (AOT) ».

2.3.1 La compilation JIT

Cette compilation « à la volée » charge le code source et le compile en code machine natif grâce à la machine virtuelle Dart. Cette méthode est utilisée pour exécuter du code dans l'interface de ligne de commande ou lorsque vous développez une application mobile afin d'utiliser des fonctionnalités telles que le débogage et le rechargement à chaud.

2.3.2 La compilation AOT

Cette compilation est effectuée en deux temps : Premièrement le code Dart est précompilé et chargé en même temps que la machine virtuelle Dart. Dans ce cas, la machine virtuelle fonctionne plus comme un système d'exécution Dart, fournissant un garbage collector et diverses méthodes natives du kit de développement logiciel (SDK) à l'application.

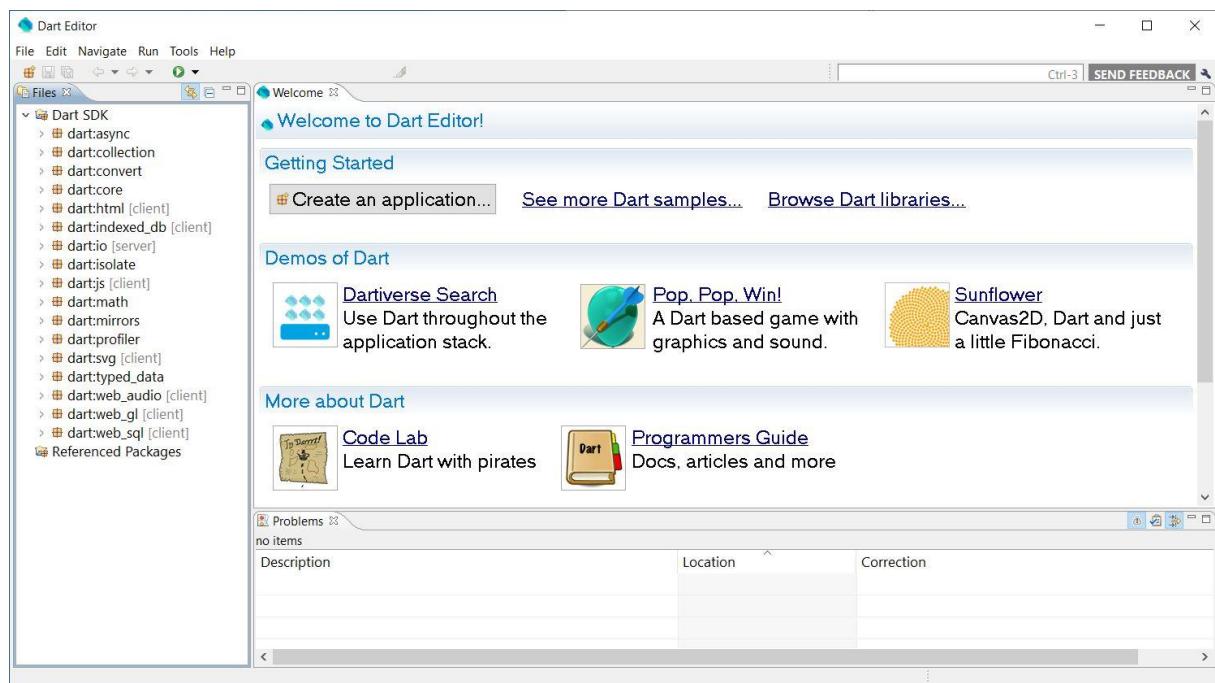
2.4 Les IDE de DART

2.4.1 Dart Editor



Le plus ancien des éditeurs pour langage Dart, publié en novembre 2011 par Google, il est un programme open source basé sur des composants Eclipse, pour les systèmes d'exploitation Mac OS X, Windows et Linux.

L'éditeur prend en charge la coloration syntaxique, la complétion de code, la compilation JavaScript, l'exécution d'applications Web, un serveur Dart et le débogage. En août 2012, Google a annoncé la sortie d'un plugin Eclipse pour le développement de Dart. En fin en avril 2015, Google a annoncé que l'éditeur Dart serait retiré au profit de l'environnement de développement intégré JetBrains (IDE).

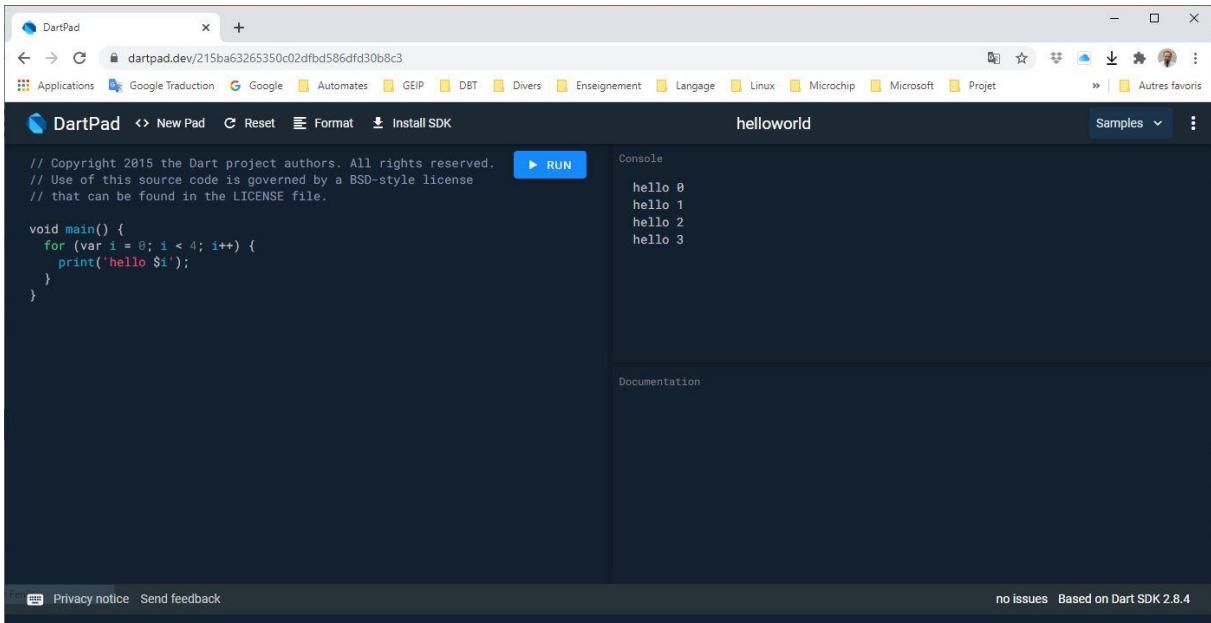


2.4.2 Dart Pad



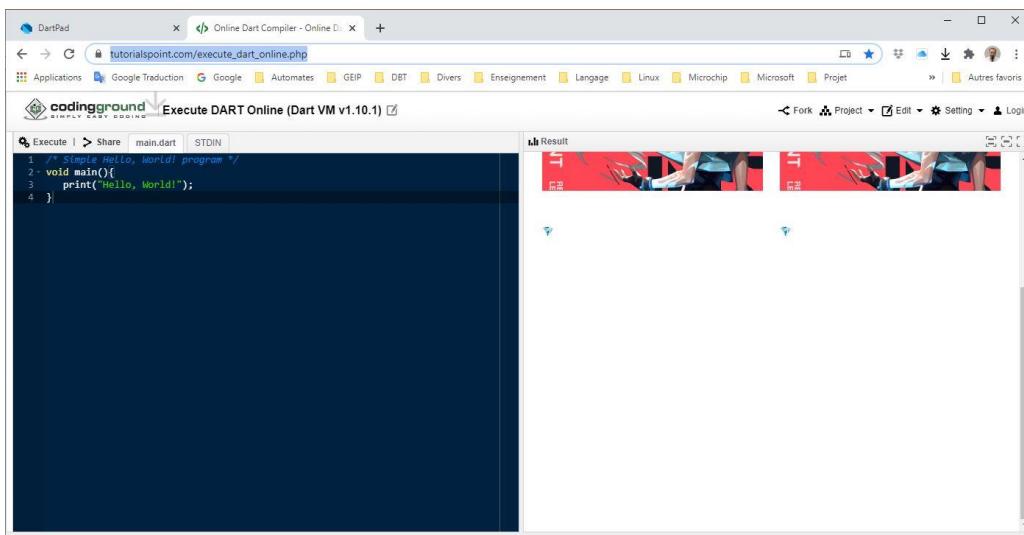
L'équipe Dart a créé DartPad début 2015, pour offrir un moyen plus simple de commencer à utiliser Dart. Il s'agit d'un éditeur entièrement en ligne à partir duquel vous pouvez expérimenter les interfaces de programmation d'application (API) Dart et exécuter du code Dart.

Il prend en charge la coloration syntaxique, l'analyse de code, l'achèvement de code, la documentation et l'édition HTML et CSS.

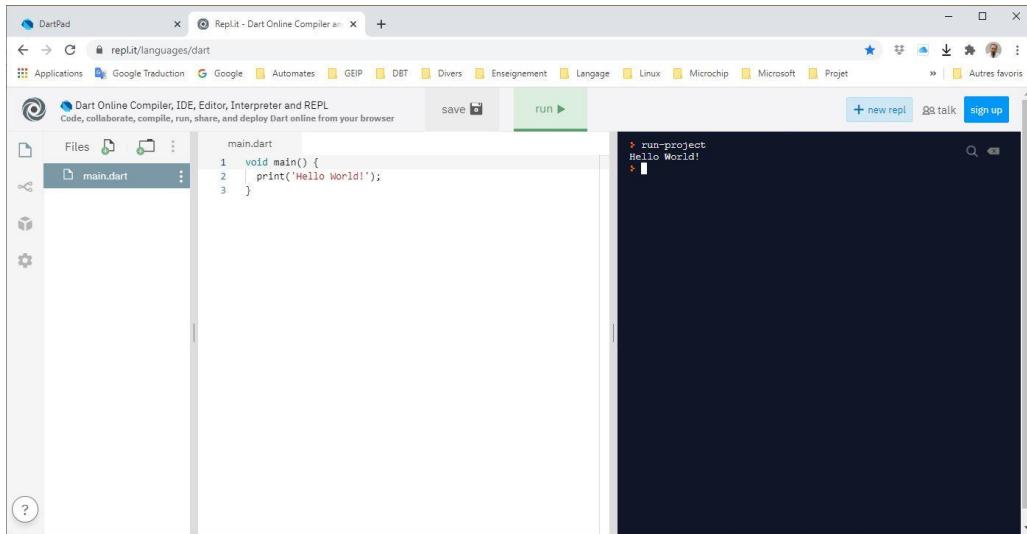


<https://dartpad.dartlang.org/>

Dart Pad a été suivi par de nombreux autres éditeurs en ligne, la liste ci-dessous n'est pas exhaustive.



https://www.tutorialspoint.com/execute_dart_online.php



<https://repl.it/languages/dart>

A screenshot of the ideone.com website. A user has submitted a Dart program named 'main.dart' with the following code: import 'dart:io'; void main() { } . The submission was successful, taking 0.94s and 118068KB. The page shows standard input and output fields, and a preview of the code with visibility settings and sharing options.

<https://ideone.com/l/dart>

A screenshot of the jdoodle.com website, specifically the 'Online Dart IDE' section. It features a code editor with a simple program that calculates the hypotenuse of a right-angled triangle. Below the editor is an 'Execute Mode, Version, Inputs & Arguments' dropdown set to '2.5.1' and an 'Interactive' button. A 'Result' panel at the bottom shows the output of the executed code.

<https://www.jdoodle.com/execute-dart-online/>

2.4.3 Plugins



Le plugin Dart, qui est maintenant l'IDE recommandé pour Google pour Dart, est disponible pour IntelliJ IDEA, PyCharm, PhpStorm et WebStorm.

Ce plugin prend en charge de nombreuses fonctionnalités telles que la coloration syntaxique, la complétion de code, l'analyse, la refactorisation, le débogage, etc.

D'autres plugins sont disponibles pour les éditeurs comme Sublime Text, Atom, Emacs, Vim et Visual Studio Code.

2.5 Mon premier programme Dart : Bonjour le monde !

La façon la plus simple de commencer à coder est d'utiliser l'outil DartPad. C'est un excellent outil en ligne pour apprendre et expérimenter les fonctionnalités linguistiques de Dart. Il prend en charge les bibliothèques principales de Dart, à l'exception des bibliothèques VM telles que dart:io.

Le premier code que nous rencontrons lorsque nous apprenons un langage est le fameux « Hello World ! ». Pour changer un peu, je vous propose d'écrire le code « Bonjour le monde ! », qui est un code Dart de base, alors jetons un coup d'œil :

```
// prog1 : Bonjour le monde

main() {                      // Point d'entrée d'un programme Dart
  var a = "monde";           // Définition et initialisation d'une variable
  var b = "Bonjour";
  print('$b le $a !');      // Appel de la fonction affichage sur la console
}
```

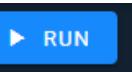
Ce code contient des fonctionnalités de base du langage :

- Chaque application Dart doit avoir une fonction de point d'entrée de niveau supérieur, c'est-à-dire fonction principale.
- Comme nous l'avons vu précédemment, bien que Dart soit de type sécurisé, les annotations de type sont facultatives. Ici, nous déclarons deux variables sans type et nous leur affectons une valeur littérale de type chaîne de caractère. Une chaîne de caractère peut être entourée de guillemets simples ou doubles, par exemple, 'Bonjour' ou "Bonjour".
- Pour afficher la sortie sur la console, nous pouvons utiliser la fonction **print()** (qui est une autre fonction de niveau supérieur). Avec la technique d'interpolation de chaîne, l'instruction **\$a** à l'intérieur d'une chaîne de caractères résout la valeur de la variable **a**. Dart appelle la méthode **toString()** de l'objet. Nous explorerons plus en détail l'interpolation de chaîne plus loin dans ce chapitre, dans la section « *Types et variables de Dart* », lorsque nous parlerons du type chaîne.
- Nous pouvons utiliser la syntaxe **//** comment pour écrire des commentaires sur une seule ligne. Dart a également des commentaires multilignes avec la syntaxe **/* commentaire */**, comme suit:

```
// ceci est un commentaire sur une seule ligne

/*
  Ceci est un long commentaire multiligne
*/
```

Nous pouvons choisir d'exécuter ce code dans le navigateur en cliquant sur le bouton



Dans ce cas, nous obtenons l'affichage suivant :

The screenshot shows the DartPad interface. In the code editor, there is a single-line Dart program:

```
// prog1 : Bonjour le monde
main() {
    var a = "monde"; // Point d'entrée d'un programme Dart
    var b = "Bonjour";
    print("$a $b !");
}
```

A blue "RUN" button is visible next to the code. To the right, under the heading "Console", the output "Bonjour le monde !" is displayed.

Mais nous avons aussi la possibilité d'exécuter ce code localement sur notre ordinateur préconfiguré avec un SDK Dart. Dans cas, nous devons enregistrer ce texte dans un fichier Dart (extension *.dart), puis l'exécuter avec la commande « dart » dans un terminal (Ex : dart prog1.dart). Cela exécutera la fonction principale du script Dart.

The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The terminal output is as follows:

```
Microsoft Windows [version 10.0.18363.900]
(c) 2019 Microsoft Corporation. Tous droits réservés.

C:\Users\Didier JUGE-HUBERT>cd \Dart\Mobile_Image

C:\Dart\Mobile_Image>cd prog1

C:\Dart\Mobile_Image\prog1>cd bin

C:\Dart\Mobile_Image\prog1\bin>dart prog1.dart
Bonjour le monde!

C:\Dart\Mobile_Image\prog1\bin>
```

Il est à noter que le type de retour de la fonction principale a été omis, donc cela suppose l'utilisation du type dynamique, que nous explorerons plus tard.

2.6 Un monde de productivité

Dart n'est pas seulement un langage, pas du moins dans son concept. Le SDK Dart est livré avec un ensemble d'outils (vu dans le paragraphe précédente sur les outils de développement Dart) dont Flutter bénéficiera pour aider avec les tâches courantes pendant la phase de développement, telles que les suivantes :

- Les compilateurs Dart JIT et AOT
- Profilage, débogage et journalisation avec Dart DevTools
- Analyse de code statique avec son analyseur intégré
<https://dart.dev/guides/language/analysis-options>

2.7 Dart un apprentissage facile

Dart est un nouveau langage pour de nombreux développeurs, et apprendre un nouveau cadre et un nouveau langage en même temps peut être difficile. Cependant, Dart simplifie cette tâche en ne réinventant pas les concepts, en les affinant simplement et en essayant de les rendre aussi efficaces que possible pour les tâches désignées.

Dart est inspiré de nombreux langages modernes et matures tels que Java, JavaScript, C #, Swift et Kotlin.



Dans cet esprit, la lecture du code Dart, même sans connaître le langage en profondeur, est possible.

Pour vous en convaincre, jetez également un œil à la page d'accueil de la documentation officielle:

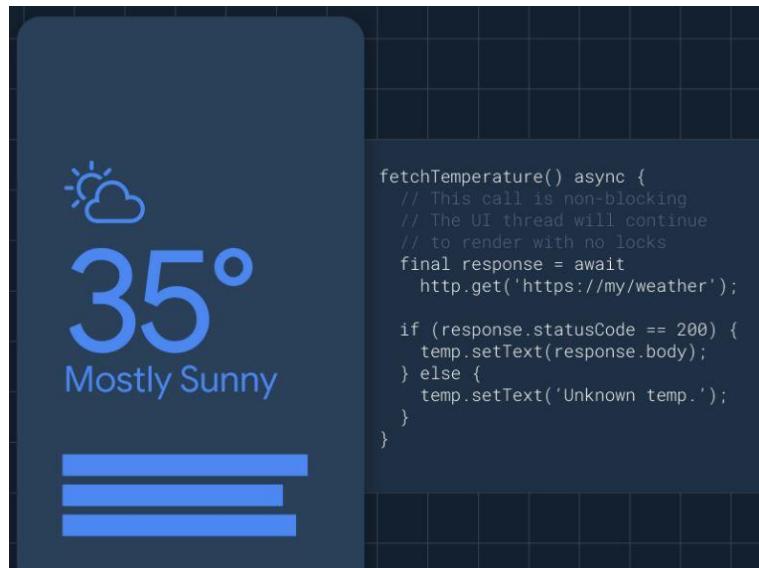
A screenshot of a web browser displaying the Dart documentation homepage. The URL in the address bar is dart.dev/documentation. The page has a dark header with the Dart logo and navigation links for Docs, Platforms, Community, Try Dart, Get Dart, and a search icon. On the left, there's a sidebar with dropdown menus for Samples & tutorials, Language, Core libraries, Packages, Development, Tools & techniques, Resources, and Related sites (which includes API reference, Blog, DartPad (online editor), Flutter, and Package site). The main content area is titled "Dart documentation" and features several cards: "Language samples" (a brief, example-based introduction to the Dart language), "Language tour" (a more thorough introduction), "Effective Dart" (best practices for building consistent, maintainable, efficient Dart code), "Library tour" (an example-based introduction to the major features in the Dart SDK's core libraries), "Dart SDK" (what's in the SDK and how to install it), and "Futures, async, await" (how to write asynchronous Dart code that uses futures and the async and await keywords). A message at the top of the content area says "Google is committed to advancing racial equity for Black communities See how.".

La documentation et les guides sont très clairs et pédagogiques. De plus, la formidable communauté aide le développeur à apprendre sans problèmes.

2.8 La maturité de Dart

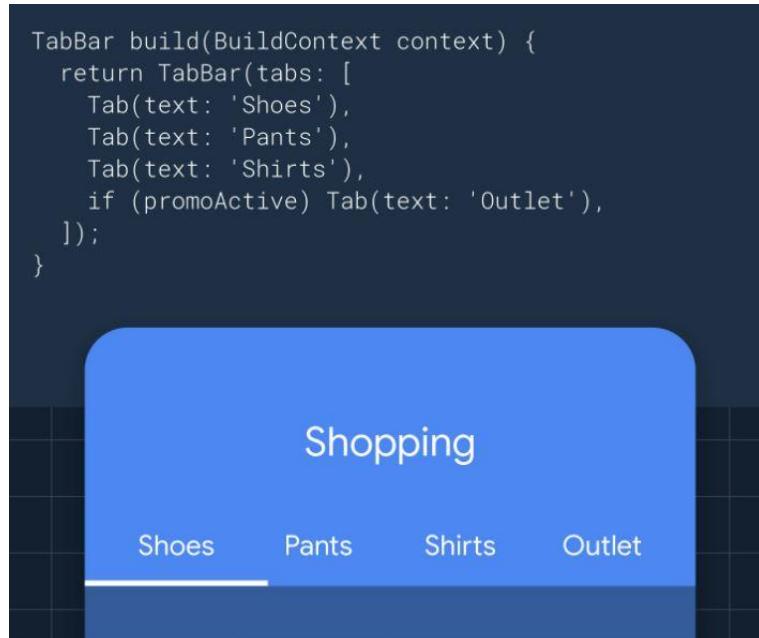
En dépit d'être un langage relativement nouveau, Dart n'est pas pauvre ou manque de ressources. Au contraire, à partir de la version 2.0, Dart dispose déjà de diverses ressources qui aident le développeur à écrire un code très performant et efficace.

Une fonctionnalité parfaite pour illustrer cela est la fonctionnalité d'attente asynchrone:



Elle permet au développeur d'écrire des appels non bloquants avec une syntaxe très simple, permettant à l'application de continuer à effectuer le rendu sans inconvénient.

Alors que Dart se concentre sur le développeur, une autre chose importante pour les développeurs mobiles et Web est la création d'interfaces utilisateur. Dans cet esprit, la syntaxe de Dart est facile à comprendre lorsque vous pensez en termes d'interface utilisateur.



Ces captures d'écran sont tirées du site officiel de Dart: <https://dart.dev/>.

En conclusion, Dart évolue aux côtés de Flutter, et ce ne sont que quelques-unes des forces importantes que le langage fournit au framework. Tant que vous réalisez que Dart est facile à apprendre et contribue à la puissance de Flutter, le défi d'apprendre une nouvelle langue avec un nouveau cadre devient plus facile et même agréable.

Dans ce cours, nous n'allons pas plonger trop profondément dans les détails de la syntaxe de Dart. Vous pouvez consulter des codes sources pour des exemples de syntaxe et l'utiliser comme guide d'étude ou comme parcours d'apprentissage pour le langage. Plus tard, vous pourrez explorer une syntaxe ou des fonctionnalités spécifiques tout en avançant dans votre parcours avec le framework Flutter.

3 Les bases du langage Dart

3.1 Introduction

Si vous connaissez déjà d'autres langages de programmation inspirés de l'ancien langage C ou si vous avez une certaine expérience de JavaScript, une grande partie de la syntaxe Dart vous sera facile à comprendre.

Dart fournit les opérateurs les plus courants pour manipuler les variables. Ses types intégrés sont les plus courants trouvés dans les langages de programmation de haut niveau, avec quelques particularités. En outre, les flux et les fonctions de contrôle sont très similaires aux formes typiques.

Nous allons passer en revue une partie de la structure du langage de programmation Dart avant de plonger dans Flutter.

Pour plus informations sur le langage Dart, vous pouvez vous référer à la visite linguistique de Dart pour un guide d'apprentissage rapide et facile sur Dart: <https://dart.dev/guides/language/tour>.

3.2 Les opérateurs Dart

Dans Dart, les opérateurs ne sont rien de plus que des méthodes définies dans des classes avec une syntaxe particulière. Ainsi, lorsque vous utilisez des opérateurs tels que `x == y`, c'est comme si vous invoquiez la méthode `x. == (y)` pour comparer l'égalité.

Nb : Comme vous l'avez peut-être noté, nous invoquons une méthode sur x, ce qui signifie que x est une instance d'une classe qui a des méthodes. Dans Dart, tout est une instance Object; tout type que vous définissez est également une instance Object.

Ce concept signifie que les opérateurs peuvent être remplacés afin que vous puissiez écrire votre propre logique. Encore une fois, si vous avez une certaine expérience en Java, C #, JavaScript ou des langages similaires, vous pouvez ignorer la plupart des opérateurs, car ils sont très similaires dans plusieurs langages.

Dart possède les opérateurs suivants:

- Arithmétique
- Incrémentation et décrémentation
- Égalité et relationnel
- Vérification et conversion de type
- Opérateurs logiques
- Manipulation de bits
- « Null-safe » et « Null-aware » (les langages de programmation modernes fournissent cet opérateur pour faciliter la gestion des valeurs nulles)

Nous allons les examiner un par un en détail.

3.2.1 Les opérateurs arithmétiques

Dart est livré avec de nombreux opérateurs typiques qui fonctionnent comme dans de nombreux langages.

Nous allons retrouver les éléments suivants:

- ⊕ + addition
- ⊕ - soustraction
- ⊕ * multiplication
- ⊕ / division simple (valeur de retour un réel)
- ⊕ ~/ division entière (valeur de retour un entier)
- ⊕ % modulo (le reste de la division entière).
- ⊕ -expression opérateur unaire de négation (qui inverse le signe de l'expression).

Certains opérateurs ont un comportement différent selon le type de l'opérande de gauche; par exemple, l'opérateur + peut être utilisé pour additionner des variables de type num, mais aussi pour concaténer des chaînes. C'est parce qu'ils ont été implémentés différemment dans les classes correspondantes.

3.2.2 Les opérateurs d'incrémantation et de décrémentation

Ceux sont également des opérateurs courants et ils sont implémentés en type nombre, comme suit:

- ⊕ ++var ou var++ pour incrémenter 1 la variable var
- ⊕ --var ou var-- pour décrémenter 1 la variable var

Une bonne application des opérateurs d'incrémantation et de décrémentation concerne les opérations de comptage sur les boucles.

3.2.3 Les opérateurs d'égalité et relationnels

Les opérateurs Dart d'égalité sont les suivants:

- ⊕ == pour vérifier si les opérandes sont égaux.
- ⊕ != pour vérifier si les opérandes sont différents.

Pour les tests relationnels, les opérateurs sont les suivants:

- ⊕ > pour vérifier si l'opérande de gauche est supérieur que l'opérande de droite.
- ⊕ < pour vérifier si l'opérande de gauche est inférieur à l'opérande de droite.
- ⊕ >= pour vérifier si l'opérande de gauche est supérieur ou égal à l'opérande de droite.
- ⊕ <= pour vérifier si l'opérande de gauche est inférieur ou égal à l'opérande de droite.

3.2.4 Les opérateurs de vérification de type et conversion

Comme vous le savez déjà, Dart est un langage où le typage est facultatif. Donc, des opérateurs de vérification de type peuvent être utiles pour vérifier les types à l'exécution :

- ⊕ is pour vérifier si l'opérande a le type testé
- ⊕ is! pour vérifier si l'opérande n'a pas le type testé

Le résultat de ces opérateurs sera différent selon le contexte de l'exécution. Dans DartPad, la sortie est vraie pour le contrôle du type double ; cela est dû à la façon dont JavaScript traite les nombres et Dart pour le Web est précompilé en JavaScript pour être exécuté sur les navigateurs Web.

3.2.5 Les opérateurs logiques

Les opérateurs logiques dans Dart sont les opérateurs courants appliqués aux opérandes booléens ; il peut s'agir de variables, d'expressions ou de conditions. De plus, ils peuvent être combinés avec des expressions complexes. Les opérateurs logiques fournis sont les suivants :

- ⊕ !expression pour nier le résultat d'une expression.
- ⊕ || pour appliquer un OU logique entre deux expressions.
- ⊕ && pour appliquer un ET logique entre deux expressions.

3.2.6 Les opérateurs de manipulation de bits

Dart fournit des opérateurs de manipulation et de décalage pour manipuler des bits de manière individuel dans une variable, généralement sur une variable de type `num`. Ces opérateurs sont les suivants :

- ⊕ & pour faire un ET logique entre les deux opérandes.
- ⊕ | pour faire un OU logique entre les deux opérandes.
- ⊕ ^ pour faire un OU logique exclusif entre les deux opérandes.
- ⊕ ~ opérande pour inverser les bits de l'opérande.
- ⊕ << pour décaler l'opérande de gauche de x bits vers la gauche (introduit des 0 à droite)
- ⊕ >> pour décaler l'opérande de gauche de x bits vers la droite (introduit des 0 à gauche)

Comme les opérateurs arithmétiques, les opérateurs de manipulation de bits ont également des opérateurs d'affectation raccourcis, et ils fonctionnent exactement de la même manière que ceux présentées précédemment (`<=>`, `&=`, `^=` et `|=`).

3.2.7 Les opérateurs « Null-safe » et « Null-aware »

Suivant la tendance des langages OOP modernes, Dart fournit une syntaxe de vérification de la valeur `Null` qui évalue et renvoie une expression en fonction de sa valeur nulle / non nulle. L'évaluation fonctionne de la manière suivante :

`expression1 ?? expression2` : si `expression1` n'est pas nulle, elle renvoie sa valeur; sinon, il évalue et renvoie la valeur de `expression2`.

En plus de l'opérateur d'affectation `=`, et de ceux répertoriés dans les opérateurs correspondants, Dart fournit également une combinaison entre l'affectation et l'expression prenant en charge les valeurs nulles; c'est-à-dire l'opérateur « `??=` », qui affecte une valeur à une variable uniquement si sa valeur actuelle est nulle.

Dart fournit également un opérateur d'accès prenant en charge les valeurs nulles « `?.` » qui empêche d'accéder aux membres `Null` d'un objet.

3.3 Les types de variables de Dart

Vous savez probablement déjà comment déclarer une variable simple, c'est-à-dire en utilisant le mot-clé `var` suivi du nom. Une chose à noter est que lorsque nous ne spécifions pas la valeur initiale de la variable, elle est supposée nulle quel que soit son type.

3.3.1 Les méthodes `final` et `const`

Si une variable ne changera pas de valeur après son affectation, vous pouvez utiliser les méthodes `final` et `const` pour déclarer celle-ci.

```
final toto;  
toto = 1;
```

La variable `toto` ne peut pas être modifiée une fois initialisée.

```
const toto = 1;
```

Tout comme le mot clé `final`, la variable `toto` ne peut pas être modifiée une fois initialisée et son initialisation doit se produire avec une déclaration.

En plus de cela, le mot clé `const` définit une constante de compilation. En tant que constante de compilation, les valeurs `const` sont connues au moment de la compilation. Ils peuvent également être utilisés pour rendre les instances d'objets ou les listes immuables, comme suit:

```
const list = const [1, 2, 3];  
const point = const Point (1,2);
```

Ceci définira la valeur des deux variables pendant la compilation, les transformant en variables complètement immuables.

3.3.2 Les types intégrés

Dart est un langage de programmation déclaratif, les types sont donc obligatoires pour les variables. Bien que les types soient obligatoires, les annotations de type sont facultatives, ce qui signifie que vous n'avez pas besoin de spécifier le type d'une variable lors de sa déclaration. Dart effectue une inférence de type.

Voici les types de données intégrés dans Dart:

- ⊕ Nombres (tels que `num`, `int` et `double`) .
- ⊕ Booléens (tels que `bool`) .
- ⊕ Collections (telles que `lists`, `arrays` et `maps`) .
- ⊕ Chaînes (`String` pour exprimer une chaîne de caractères Unicode).

3.3.2.1 Les nombres

Dart représente les nombres de deux manières :

- ⊕ `int` valeurs entières non fractionnaires signées sur 64 bits telles que -2^{63} à $2^{63}-1$.
- ⊕ `double` valeurs numériques fractionnaires avec un nombre à virgule flottante double précision de 64 bits.

Ces deux types étendent le type `num`. De plus, nous avons de nombreuses fonctions pratiques dans la bibliothèque `dart:math` pour vous aider dans les calculs.

Dart a également le type `BigInt` pour représenter des entiers de précision arbitraires, ce qui signifie que la limite de taille est la RAM de l'ordinateur en cours d'exécution. Ce type peut être très utile selon le contexte ; cependant, il n'a pas les mêmes performances que les types `num` et vous devez en tenir compte lorsque vous décidez de l'utiliser.

Maintenant, vous pouvez envisager de placer `BigInt` partout où vous utiliseriez des entiers pour éviter les débordements, mais rappelez-vous que `BigInt` n'a pas les mêmes performances que les types `int`, ce qui le rend inadapté à tous les contextes.

3.3.2.2 Les booléens

Dart fournit les deux valeurs littérales bien connues pour le type `bool` : `true` et `false`.

Les types booléens sont des valeurs de vérité simples qui peuvent être utiles pour n'importe quelle logique. Une chose que vous avez peut-être remarquée, mais que je veux renforcer, concerne les expressions. Nous savons déjà que les opérateurs, tels que par exemple `>` ou `==`, ne sont rien de plus que des méthodes avec une syntaxe spéciale définie dans les classes, et, bien sûr, ils ont une valeur de retour qui peut être évaluée dans des conditions. Ainsi, le type de retour de toutes ces expressions est `bool` et, comme vous le savez déjà, les expressions booléennes sont importantes dans tout langage de programmation.

3.3.2.3 Les collections

Dans Dart, les listes sont considérées comme des tableaux dans d'autres langages de programmation avec quelques méthodes pratiques pour les manipuler.

Les listes ont l'opérateur `[index]` pour accéder aux éléments à l'index donné et, en plus, l'opérateur `+` peut être utilisé pour concaténer deux listes en renvoyant une nouvelle liste avec l'opérande de gauche suivi de l'opérande de droite.

Une autre chose importante à propos des listes Dart est la contrainte de longueur. C'est ainsi que nous définissons les listes précédentes, en les faisant croître selon les besoins en utilisant la méthode `add`, qui l'agrandira pour ajouter l'élément.

Une autre façon de définir la liste consiste à définir sa longueur lors de sa création. Les listes de taille fixe ne peuvent pas être développées, il est donc de la responsabilité du développeur de savoir où et quand utiliser des listes de taille fixe, car cela peut lever des exceptions si vous essayez d'ajouter ou d'accéder à des éléments non valides.

Les `maps` Dart sont des collections dynamiques permettant de stocker des valeurs sur une base de clé, où la récupération et la modification d'une valeur sont toujours effectuées à l'aide de sa clé associée. La clé et la valeur peuvent avoir n'importe quel type; si nous ne spécifions pas les types clé-valeur, ils seront déduits par Dart comme `Map<dynamic, dynamic>`, avec ses clés et ses valeurs de type dynamique.

3.3.2.4 Les chaînes de caractères

Dans Dart, les chaînes sont une séquence de caractères (code UTF-16) qui sont principalement utilisées pour représenter du texte. Les chaînes Dart peuvent être des lignes simples ou multiples. Vous pouvez faire correspondre des guillemets simples ou doubles (généralement pour des lignes simples) et des chaînes multilignes en faisant correspondre des guillemets triples.

Vous pouvez utiliser l'opérateur `+` pour concaténer des chaînes. Le type `String` implémente des opérateurs utiles autres que le plus `(+)`. Il implémente l'opérateur munitoplateur `(*)` où la chaîne est répétée un nombre spécifié de fois, et l'opérateur `[index]` récupère le caractère à la position `index` spécifiée.

Dart a une syntaxe utile pour interrober la valeur des expressions Dart dans les chaînes: `$ {}`, qui fonctionne comme suit:

```

void main() {
  String uneChaine = "Ceci est une chaîne";
  print ("La valeur de la chaîne est: $uneChaine");
  // affiche => La valeur de la chaîne est: Ceci est une chaîne
  print ("La longueur de la chaîne est: ${uneChaine.length}");
  // affiche => La longueur de la chaîne est: 19
}

```

▶ RUN

Console

```

La valeur de la chaîne est: Ceci est une chaîne
La longueur de la chaîne est: 19

```

Comme vous l'avez peut-être remarqué, lorsque nous n'insérons qu'une variable et non une valeur d'expression dans la chaîne, nous pouvons omettre les accolades et simplement ajouter directement **\$variable**.

3.3.2.5 Les valeurs littérales

Vous pouvez utiliser les syntaxes `[]` et `{}` pour initialiser respectivement des variables telles que des **List** et des **Map**. Voici quelques exemples fournis par le langage Dart pour créer des objets de types intégrés :

Type	Exemple de valeurs littérales
<code>int</code>	10, 1, -1, 5 et 0
<code>double</code>	10.1, 1.2, 3.123 et -1.2
<code>bool</code>	true et false
<code>String</code>	"Dart", "Dash" et "" "multiligne String" ""
<code>List</code>	[1,2,3] and ["one", "two", "three"]
<code>Map</code>	{"key1": "val1", "b": 2}

3.3.3 L'inférence de type

Dans les exemples précédents, nous avons montré deux façons de déclarer des variables : en utilisant le type de la variable, comme `int` et `String`, ou en utilisant le mot-clé `var`. Donc, maintenant vous vous demandez peut-être comment Dart sait de quel type de variable il s'agit si vous ne le spécifiez pas dans une déclaration.

À partir de la documentation Dart (<https://dart.dev/guides/language/effective-dart/documentation>), considérez la sentence suivante :

"L'analyseur peut déduire des types pour les champs, les méthodes, les variables locales et la plupart des arguments de type générique. Lorsque l'analyseur ne dispose pas de suffisamment d'informations pour déduire un type spécifique, il utilise le type dynamique."

Cela signifie que, lorsque vous déclarez une variable, l'analyseur Dart déduit le type en fonction de la valeur littérale ou du constructeur d'objet. Voici un exemple (qui ne fonctionne pas sous DartPad) :

```

import 'dart:mirrors';

main() {
  var unEntier = 1;
  print(reflect(unEntier).type.reflectedType.toString());
  // affiche => int
}

```

Comme vous pouvez le voir, dans cet exemple, nous n'avons que le mot-clé `var`. Nous n'avons spécifié aucun type, mais comme nous avons utilisé la valeur littérale (1), l'outil d'analyse peut déduire le type

avec succès. Les variables locales obtiennent le type déduit par l'analyseur lors de l'initialisation. Dans l'exemple précédent, essayer d'attribuer une valeur de type chaîne à unEntier échouerait.

Alors, considérons le code suivant :

```
main() {
  var a; // la variable n'est pas initialisée donc de type dynamic
  a = 1; // maintenant a est de type int
  a = "a"; // maintenant a est de type String
  print(a is int); // affiche false
  print(a is String); // affiche true
  print(a is dynamic); // affiche true
  print(a.runtimeType); // affiche String
}
```

Comme vous l'avez peut-être remarqué, `a` est un type `String` et un type `dynamic`. `dynamic` est un type spécial et il peut prendre n'importe quel type au moment de l'exécution; par conséquent, toute valeur peut également être convertie en `dynamic`. Dart peut déduire le type pour les champs, les retours de méthode et les arguments de type générique.

3.4 Les instructions et les fonctions

3.4.1 Les traitements conditionnels et les boucles

Vous avez vu comment utiliser les variables et les opérateurs Dart pour créer des expressions conditionnelles. Pour travailler avec des variables et des opérateurs, nous devons généralement implémenter des traitements conditionnels pour que notre code Dart prenne la direction appropriée dans notre logique.

Dart fournit une syntaxe qui est très similaire à d'autres langages de programmation qui est comme suit :

- ✚ `if-else`
- ✚ `switch/case`
- ✚ Bouclage avec `for`, `while` et `do-while`
- ✚ `break` et `continue`
- ✚ `asserts`
- ✚ Exceptions avec `try / catch` and `throw`

La syntaxe Dart n'a pas de particularités importantes à revoir en détail. Veuillez-vous référer à la documentation sur les traitements conditionnels pour plus de détails: <https://dart.dev/guides/language/language-tour#control-flow-statements>.

3.4.2 Les fonctions

Dans Dart, `Function` est un type, comme `String` ou `num`. Cela signifie qu'ils peuvent également être affectés à des champs ou à des variables locales, ou transmis en tant que paramètres à d'autres fonctions. Prenons l'exemple suivant :



```
String direBonjour() {
    return "Bonjour le monde!";
}

void main() {
    var direBonjourFonction = direBonjour;
    // assigne la fonction à une variable
    print(direBonjourFonction()); // affiche Bonjour le monde!
}
```

Dans cet exemple, la variable `direBonjourFonction` stocke la fonction `direBonjour` elle-même et ne l'appelle pas. Plus tard, nous pouvons l'invoquer en ajoutant `()` au nom de la variable comme s'il s'agissait d'une fonction.

Si vous essayez d'appeler une variable sans fonction peut entraîner une erreur du compilateur.

Le type de retour de fonction peut également être omis, de sorte que l'analyseur Dart déduit le type à partir de l'instruction `return`. Si aucune instruction `return` n'est fournie, elle suppose `return null`. Si vous voulez lui dire qu'il n'a pas de retour, vous devez le marquer comme `void`.

La fonction `direBonjour` aurait pu s'écrire de la manière suivante :

```
direBonjour() {
    return "Bonjour le monde!";
}
```

Une autre façon d'écrire cette fonction consiste à utiliser la syntaxe abrégée,

```
() => expression ;
```

qui est également appelée fonction « Flèche » ou fonction « Lambda ». Dans notre exemple cela donnerait :

```
direBonjour() => "Bonjour le monde!" ;
```

Vous ne pouvez pas écrire d'instructions à la place d'une expression, mais vous pouvez utiliser les expressions conditionnelles déjà connues (c'est-à-dire `? : ou ??`).

3.4.2.1 Les paramètres des fonctions

Une fonction peut avoir deux types de paramètres : facultatifs et obligatoires. De plus, comme avec la plupart des langages de programmation modernes, ces paramètres peuvent être nommés sur appel pour rendre le code plus lisible.

Le type du paramètre n'a pas besoin d'être spécifié; dans ce cas, le paramètre prend le type dynamique:

3.4.2.1.1 Les paramètres obligatoires

Cette simple définition de fonction avec des paramètres est obtenue en les définissant simplement de la même manière que la plupart des autres langages. Dans la fonction suivante, `nom` et `message` sont des paramètres obligatoires, donc l'appelant doit les transmettre lors de son appel:

The screenshot shows a Dart code editor interface. On the left, there is a code editor window containing the following code:

```
direBonjour(String nom, String message) => "Bonjour $nom. $message.";  
void main() {  
  var direBonjourFonction = direBonjour;  
  // assigne la fonction à une variable  
  print(direBonjourFonction("Didier", "Bonjour à tous"));  
}
```

On the right, there is a "RUN" button and a "Console" output window. The console output is:

```
▶ RUN  
Console  
Bonjour Didier. Bonjour à tous.
```

3.4.2.1.2 Les paramètres facultatifs

Parfois, tous les paramètres ne sont pas nécessaires pour une fonction, elle peut donc également définir des paramètres facultatifs. Les paramètres facultatifs sont de deux types :

- Paramètres positionnés facultatifs
- Paramètres nommés facultatifs

La définition de paramètres positionnés facultatifs est effectuée à l'aide de la syntaxe `[]`. Les paramètres positionnés facultatifs doivent être placés après tous les paramètres requis, comme suit :

The screenshot shows a Dart code editor interface. On the left, there is a code editor window containing the following code:

```
direBonjour (String nom, [String message]) => "Bonjour $nom. $message.";  
void main() {  
  var direBonjourFonction = direBonjour;  
  // assigne la fonction à une variable  
  print(direBonjourFonction("Didier"));  
}
```

On the right, there is a "RUN" button and a "Console" output window. The console output is:

```
▶ RUN  
Console  
Bonjour Didier. null.
```

Quand vous exécutez le code précédent sans passer une valeur pour `message`, vous verrez `null` à la fin de la chaîne renournée.

The screenshot shows a Dart code editor interface. On the left, there is a code editor window containing the following code:

```
direBonjour (String nom, [String message]) => "Bonjour $nom. $message.";  
void main() {  
  print(direBonjour('Mon amis')); // affiche Bonjour mon amis, null.  
  print(direBonjour('Mon amis', 'Comment vas-tu ?'));  
  // affiche Bonjour mon amis. Comment vas-tu?.  
}
```

On the right, there is a "RUN" button and a "Console" output window. The console output is:

```
▶ RUN  
Console  
Bonjour Mon amis. null.  
Bonjour Mon amis. Comment vas-tu ?.
```

Lorsque le paramètre facultatif n'est pas spécifié, la valeur par défaut est nulle, sauf si vous spécifiez des valeurs par défaut pour les paramètres facultatifs. Pour définir une valeur par défaut pour un paramètre facultatif, vous devez l'ajouter après un signe `=` juste après la définition du paramètre. Ne pas spécifier le paramètre entraîne l'impression du message par défaut, comme suit:

The screenshot shows a Dart code editor interface. On the left, there is a code editor window containing the following code:

```
direBonjour (String nom, [String message = "Bienvenue dans le monde de Dart"]) =>  
  "Bonjour $nom. $message.";  
void main() {  
  print(direBonjour('Mon amis')); // affiche Bonjour mon amis, null.  
  print(direBonjour('Mon amis', 'Comment vas-tu ?'));  
  // affiche Bonjour mon amis. Comment vas-tu?.  
}
```

On the right, there is a "RUN" button and a "Console" output window. The console output is:

```
▶ RUN  
Console  
Bonjour Mon amis. Bienvenue dans le monde de Dart.  
Bonjour Mon amis. Comment vas-tu ?.
```

Les paramètres nommés facultatifs sont définis à l'aide de la syntaxe {}. Ils doivent également aller après tous les paramètres requis. L'appelant doit spécifier le nom du paramètre nommé facultatif, comme suit :

The screenshot shows a Dart code editor with a blue 'RUN' button. The code defines a function `direBonjour` that takes two parameters: `String nom` and `{String message}`. It prints "Bonjour \$nom. \$message.". In the `main` function, it calls `print(direBonjour('Mon amis'))` and `print(direBonjour('Mon amis', message:'Comment vas-tu ?'))`. The output in the 'Console' window shows two lines: "Bonjour Mon amis. null." and "Bonjour Mon amis. Comment vas-tu ?".

```
direBonjour (String nom, {String message}) => "Bonjour $nom. $message.";

void main() {
  print(direBonjour('Mon amis')); // affiche Bonjour mon amis, null.
  print(direBonjour('Mon amis', message:"Comment vas-tu ?"));
  // affiche Bonjour mon amis. Comment vas-tu?.
```

3.4.3 Les fonctions anonymes

Les fonctions Dart sont des objets et elles peuvent être passées en tant que paramètres à d'autres fonctions. Nous l'avons déjà vu lors de l'utilisation de la fonction itération `forEach()`. Une fonction anonyme est une fonction qui n'a pas de nom ; il est également appelé « lambda » ou « fermeture ». La fonction `forEach()` en est un bon exemple ; nous devons lui passer une fonction qui sera exécutée avec chacun des éléments de la liste :

The screenshot shows a Dart code editor with a blue 'RUN' button. The code defines a `main` function that creates a list [1, 2, 3, 4] and iterates over it using `list.forEach((nombre) => print('Bonjour $nombre'))`. The output in the 'Console' window shows four lines: "Bonjour 1", "Bonjour 2", "Bonjour 3", and "Bonjour 4".

```
void main() {
  var list = [1, 2, 3, 4];
  list.forEach((nombre) => print('Bonjour $nombre'));
}
```

Notre fonction anonyme reçoit un élément mais ne spécifie pas de type, puis, elle affiche simplement la valeur reçue par le paramètre.

3.4.4 La portée des éléments

La portée de Dart est déterminée par la disposition du code en utilisant des accolades comme de nombreux langages de programmation. Les fonctions internes peuvent accéder aux variables jusqu'au niveau global :

The screenshot shows a Dart code editor with a blue 'RUN' button. The code defines three functions: `fonctionSimple`, `fonctionGlobale`, and `main`. `fonctionSimple` contains a call to `fonctionGlobale()`. `fonctionGlobale` contains a call to `fonctionSimple()`. In the `main` function, both `fonctionSimple` and `fonctionGlobale` are called. The output in the 'Console' window shows three lines: "fonction simple", "Pas un vrai fonction globale", and "Vrai fonction globale de niveau haut".

```
fonctionGlobale() {
  print("Vrai fonction globale de niveau haut");
}

fonctionSimple() {
  print("Fonction simple");
  fonctionGlobale() {
    print("Pas un vrai fonction globale");
  }
  fonctionGlobale();
}

main() {
  fonctionSimple();
  fonctionGlobale();
}
```

Si vous examinez le code précédent, la fonction `fonctionGlobale` de `fonctionSimple` sera utilisée à la place de la version globale, car elle est définie localement sur sa portée. Dans la fonction `main`, en revanche, la version globale de la fonction `fonctionGlobale` est utilisée, car, dans cette portée, la fonction `fonctionGlobale` interne à la `fonctionSimple` n'est pas définie.

3.5 Structures de données, collections et génériques

Dart fournit plusieurs types de structures pour manipuler un ensemble de valeurs. Les listes de Dart sont largement utilisées même dans les cas d'utilisation les plus simples.

Les génériques sont un concept lorsque vous travaillez avec des collections de données liées à un type spécifique, tel que `List` ou `Map`, par exemple. Ils garantissent qu'une collection aura des valeurs homogènes en spécifiant le type de données qu'elle peut contenir.

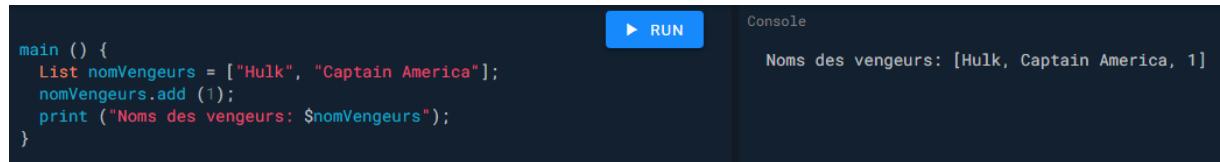
3.5.1 Comment définir les génériques

La syntaxe `< . . >` est utilisée pour spécifier le type pris en charge par une collection. Si vous regardez les exemples précédents de `Lists` et de `maps`, vous remarquerez que nous n'avons spécifié aucun type. En effet, ils sont facultatifs et Dart peut déduire le type en fonction des éléments lors de l'initialisation de la collection.

3.5.2 Quand et pourquoi utiliser des génériques

L'utilisation de génériques peut aider un développeur à maintenir et à garder le comportement de la collection sous contrôle. Lorsque nous utilisons une collection sans spécifier les types d'élément autorisés, il est de notre responsabilité d'insérer correctement les éléments. Cela, dans un contexte plus large, peut devenir coûteux, car nous devons implémenter des validations pour éviter les mauvaises insertions et pour le documenter pour une équipe.

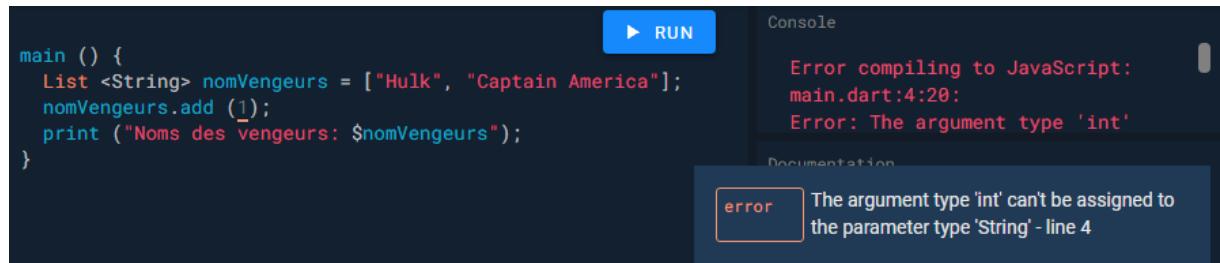
Dans l'exemple ci-dessous, comme nous avons nommé la variable `nomVengeurs`, nous nous attendons à ce que ce soit une liste de noms et rien d'autre. Malheureusement, sous forme codée, nous pouvons également insérer un nombre dans la liste, ce qui provoque une désorganisation ou une confusion :



```
main () {  
  List nomVengeurs = ["Hulk", "Captain America"];  
  nomVengeurs.add (1);  
  print ("Noms des vengeurs: $nomVengeurs");  
}
```

The screenshot shows a Dart code editor with a blue 'RUN' button. To the right is a 'Console' window displaying the output: 'Noms des vengeurs: [Hulk, Captain America, 1]'. This demonstrates that the list can contain both strings and integers.

Cependant, si nous spécifions le type `String` pour la liste, alors ce code ne se compilera pas, évitant cette confusion :



```
main () {  
  List <String> nomVengeurs = ["Hulk", "Captain America"];  
  nomVengeurs.add (1);  
  print ("Noms des vengeurs: $nomVengeurs");  
}
```

The screenshot shows a Dart code editor with a blue 'RUN' button. To the right is a 'Console' window with an error message: 'Error compiling to JavaScript: main.dart:4:20: Error: The argument type 'int' can't be assigned to the parameter type 'String' - line 4'. Below the console is a tooltip box with the text: 'error The argument type 'int' can't be assigned to the parameter type 'String' - line 4'.

3.5.3 Génériques et valeurs littérales

Si vous consultez les exemples de `List` et `Map`, vous verrez que nous avons utilisé `[]` et `{}` pour les initialiser. Avec les génériques, nous pouvons spécifier un type lors de l'initialisation, en ajoutant un préfixe `<elementType>[]` pour `List` et `<keyType, elementType>{}` pour `Map`.



```
main() {  
  var nomVengeurs = <String>["Hulk", "Spider Man", "Captain America"];  
  var citationVengeurs = <String, String>{  
    "Captain America": "Je peux faire ça toute la journée!",  
    "Spider Man": "Suis-je un vengeur?",  
    "Hulk": "Smaaaaaash!"  
};  
  
  nomVengeurs.forEach((nom) => print("$nom => " + citationVengeurs[nom].toString()));  
}
```

The screenshot shows a Dart code editor with a blue 'RUN' button. To the right is a 'Console' window displaying the output: 'Hulk => Smaaaaaash!', 'Spider Man => Suis-je un vengeur?', and 'Captain America => Je peux faire ça toute la journée!'. This demonstrates how generic types can be used to ensure consistency in data structures.

Spécifier le type pour List, dans ce cas, semble être redondant car l'analyseur Dart déduira le type de chaîne à partir des valeurs littérales que nous avons fournis. Cependant, dans certains cas, cela est important, par exemple lorsque nous initialisons une collection vide,

```
var tableauchainevide = <String> [];
```

Si nous n'avons pas spécifié le type de la collection vide, elle pourrait contenir n'importe quel type de données car elle n'inférerait pas le type générique à adopter.

3.6 Introduction à la POO dans Dart

Dans Dart, tout est objet, y compris les types intégrés. Lors de la définition d'une nouvelle classe, même si vous n'étendez rien, ce sera un descendant d'un objet. Dart le fait implicitement pour vous.

Dart est un véritable langage orienté objet. Même les fonctions sont des objets, ce qui signifie que vous pouvez effectuer les opérations suivantes :

- Attribuer une fonction en tant que valeur d'une variable.
- Passez-la comme argument à une autre fonction.
- Renvoyez-la comme résultat d'une fonction comme vous le feriez avec tout autre type, tel que `String` ou `int`.

Ceci est connu comme ayant des fonctions de première classe car elles sont traitées de la même manière que les autres types.

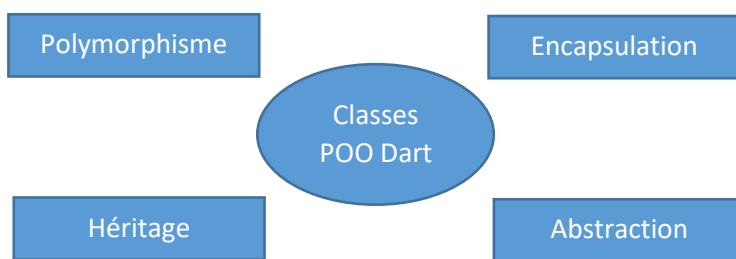
Un autre point important à noter est que Dart prend en charge l'héritage unique sur une classe, similaire à Java et à la plupart des autres langages, ce qui signifie qu'une classe ne peut hériter directement que d'une seule classe à la fois.

Voici les principaux points de la POO présents dans le langage Dart :

- Classe : il s'agit d'un moyen pour créer un objet.
- Interface : il s'agit d'une définition de contrat avec un ensemble de méthodes disponibles sur un objet. Bien qu'il n'y ait pas de type d'interface explicite dans Dart, nous pouvons atteindre l'objectif d'interface avec des classes abstraites.
- Classe énumérée : il s'agit d'un type spécial de classe qui définit un ensemble de valeurs constantes communes.
- Mixin : c'est une façon de réutiliser le code d'une classe dans plusieurs hiérarchies de classes.

3.6.1 Fonctionnalités de la POO de Dart

Chaque langage de programmation peut fournir les paradigmes de la POO à sa manière, avec une prise en charge partielle ou totale, en appliquant certains ou tous les principes suivants.



Dart applique de nombreux principes avec de nombreuses particularités qui renforcent les techniques et structures POO disponibles pour utiliser les paradigmes POO dans le langage Dart.

3.6.1.1 *Objets et classes*

Le point de départ de la POO, les objets, sont des instances de classes définies. Dans Dart, comme cela a déjà été souligné, tout est objet, c'est-à-dire que chaque valeur que nous pouvons stocker dans une variable est une instance d'une classe.

En outre, tous les objets étendent également la classe `Object`, directement ou indirectement:

- Les classes Dart peuvent avoir à la fois des membres d'instance (méthodes et champs) et des membres de classe (méthodes et champs statiques).
- Les classes Dart ne prennent pas en charge la surcharge du constructeur, mais vous pouvez utiliser les spécifications d'argument de fonction flexible du langage (facultatif, positionnel et nommé) pour fournir différentes manières d'instancier une classe. De plus, vous pouvez avoir des constructeurs nommés pour définir des alternatives.

3.6.1.2 *Encapsulation*

Dart ne contient pas explicitement de restrictions d'accès, comme les célèbres mots-clés utilisés en Java – `protected`, `private` et `public`. Dans Dart, l'encapsulation se produit au niveau de la bibliothèque plutôt qu'au niveau de la classe.

Ce qui suit s'applique également :

- Dart crée des méthodes implicites pour lire et écrire tous les champs d'une classe, afin que vous puissiez définir comment les données sont accessibles aux consommateurs et comment elles changent.
- Dans Dart, si un identificateur (classe, membre de classe, fonction de niveau supérieur ou variable) commence par un underscore (_), il est privé pour sa bibliothèque.

3.6.1.3 *Héritage et composition*

L'héritage nous permet d'étendre un objet à des versions spécialisées d'un type abstrait. Dans Dart, en déclarant simplement une classe, nous étendons déjà implicitement le type `Object`.

Ce qui suit s'applique également :

- Dart autorise l'héritage direct unique.
- Dart a un support spécial pour les « mixins », qui peut être utilisé pour étendre les fonctionnalités de classe sans héritage direct, simuler plusieurs héritages et réutiliser du code.
- Dart ne contient pas de directive de classe finale comme les autres langages ; en d'autres termes, une classe peut toujours être étendue (avoir des enfants).

3.6.1.4 *Abstraction*

Après l'héritage, l'abstraction est le processus par lequel nous définissons un type et ses caractéristiques essentielles, passant aux types spécialisés des types parents.

Ce qui suit s'applique également :

- Dart contient des classes abstraites qui permettent une définition de ce que fait/fournit quelque chose, sans se soucier de la façon dont cela est implémenté.
- Dart a le puissant concept d'interface implicite, qui fait également de chaque classe une interface, lui permettant d'être implémenté par d'autres sans l'étendre.

3.6.1.5 Polymorphisme

Le polymorphisme est obtenu par héritage et peut être considéré comme la capacité d'un objet à se comporter comme un autre. Par exemple, le type `int` est également un type `num`. Ce qui suit s'applique également :

- Dart permet de remplacer les méthodes parentes pour modifier leur comportement d'origine.
- Dart n'autorise pas la surcharge comme vous le savez peut-être. Vous ne pouvez pas définir la même méthode deux fois avec des arguments différents. Vous pouvez simuler une surcharge en utilisant des définitions d'argument flexibles ou ne pas l'utiliser du tout.

4 Les aspects avancés du langage Dart

4.1 Introduction

Vous allez voir dans ce chapitre le concept de base des objets dans Dart, par exemple, comment créer du code orienté objet dans Dart en utilisant ses concepts, tels que les interfaces, les interfaces implicites et les classes abstraites, ainsi que les mixins, pour ajouter un comportement à une classe.

Si vous êtes un programmeur expérimenté ou déjà familiarisé avec Java ou des langages similaires, cela ne sera que des révisions, car il présente de nombreuses similitudes avec les concepts typiques de la POO, tels que l'héritage et l'encapsulation. Mais je pense que certains concepts en particulier sont importants à vérifier, même si vous êtes déjà familiarisé avec la majorité des fonctionnalités de la POO, telles que les interfaces implicites et les mixins, car elles peuvent vous présenter de nouveaux concepts.

Vous apprendrez également à utiliser des bibliothèques tierces pour accélérer le développement d'un projet, à comprendre les fonctionnalités avancées du langage Dart pour commencer à développer des applications multithreading à l'aide de callbacks et de futures, et apprendre à tester votre code Dart.

4.2 Les classes Dart

Les classes Dart sont déclarées à l'aide du mot-clé `class`, suivi du nom de la classe, des classes ancêtres et des interfaces implémentées. Ensuite, le corps de la classe est entouré par une paire d'accolades, dans lesquelles vous pouvez ajouter des membres de classe, qui incluent les éléments suivants :

Champs : il s'agit de variables utilisées pour définir les données qu'un objet peut contenir.

Accesseurs : il s'agit de méthodes permettant de lire ou d'écrire les champs d'une classe, où `get` est utilisé pour récupérer une valeur, et `set` est utilisé pour modifier la valeur correspondante.

Constructeur : il s'agit de la méthode de création d'une classe où les champs d'instance d'objet sont initialisés.

Méthodes : Le comportement d'une classe objet est défini par les actions qu'il peut entreprendre. Ce sont les fonctions de .

Regardez l'exemple ci-dessous :

```
class Personne {
  String prenom;
  String nom;
  String getNomCompleter() => "$prenom $nom";
}

main () {
  Personne unePersonne = new Personne();
  unePersonne.prenom = "Clark";
  unePersonne.nom = "Kent";
  print (unePersonne.getNomCompleter()); // affiche Clark Kent
}
```

Si nous regardons la classe Personne déclarée dans le code précédent et faisons quelques observations :

- Pour instancier une classe, nous utilisons le mot clé new (optionnel), suivi de l'appel du constructeur. Au fur et à mesure que nous avançons dans votre apprentissage, vous remarquerez que ce mot-clé est moins utilisé.
- Il n'a pas de classe parent explicitement déclarée, mais il en a une, le type d'objet, comme déjà mentionné, et cet héritage se produit implicitement dans Dart.
- Il a deux champs, prenom et nom, et une méthode, `getNomComplet()`, qui concatène les deux à l'aide d'une interpolation de chaîne, puis renvoie les données.
- Il n'a aucun accesseur déclaré, alors comment avons-nous accédé à `prenom` et `nom` pour les écrire ? Un accesseur get / set par défaut est défini pour chaque champ d'une classe.
- La notation dot `class.member` est utilisée pour accéder à un membre de classe, quel qu'il soit - une méthode ou un champ.
- Nous n'avons pas défini de constructeur pour la classe, mais, comme vous le pensez peut-être, il existe un constructeur vide par défaut (sans arguments) déjà fourni.

4.3 Le type enum

Le type enum est un type courant utilisé par la plupart des langages pour représenter un ensemble de valeurs constantes finies. Dans Dart, ce n'est pas différent. En utilisant le mot clé `enum`, suivi des valeurs constantes, vous pouvez définir un type `enum` :

```
enum TypePersonne {
    etudiant, employe
}
```

Notez que vous ne définissez que les noms de valeur. Les types `enum` sont des types spéciaux avec un ensemble de valeurs finies qui ont une propriété d'index représentant sa valeur. Voyons maintenant comment cela fonctionne. Tout d'abord, nous ajoutons un champ à notre classe Personne précédemment définie pour stocker son type:

```
class Personne {
    String prenom;
    String nom;
    TypePersonne type;
    String getNomComplet() => "$prenom $nom";
}
```

Ensuite, nous pouvons l'utiliser comme n'importe quel autre champ:

```
main() {
  print(TypePersonne.values);
  Personne unePersonne = new Personne();
  unePersonne.type = TypePersonne.employe;
  unePersonne.prenom = "Clark";
  unePersonne.nom = "Kent";
  print(unePersonne.type); // affiche TypePersonne.employe
  print(unePersonne.type.index); // affiche 1
}
```

▶ RUN

Console

```
[TypePersonne.etudiant, TypePersonne.employe]
TypePersonne.employe
1
```

Vous pouvez voir que la propriété `index` est égale à zéro, en fonction de la position de déclaration de la valeur. En outre, vous pouvez voir que nous appelons directement la lecture des valeurs sur l'énumération `TypePersonne`. Il s'agit d'un membre statique du type `enum` qui renvoie simplement une liste avec toutes ses valeurs.

4.4 La notation en cascade

Nous avons vu que Dart fournit la notation par points pour accéder à un membre de classe. En plus de cela, nous pouvons également utiliser la notation double point/cascade, qui nous permet d'enchaîner une séquence d'opérations sur le même objet :

The screenshot shows a code editor interface with a dark theme. On the left, there is a snippet of Dart code:

```
enum TypePersonne {
  etudiant, employe
}

class Personne {
  String prenom;
  String nom;
  TypePersonne type;
  String getNomComplet() => "$prenom $nom";
}

main () {
  Personne unePersonne = new Personne()
  ..prenom = "Clark"
  ..nom = "Kent";
  print (unePersonne.getNomComplet()); // affiche Clark Kent
}
```

In the top right corner, there is a blue button labeled "RUN". To the right of the code editor, there is a "Console" window showing the output:

```
Console
Clark Kent
```

Le résultat est le même que lors de l'utilisation de l'approche typique. C'est juste un bon moyen d'écrire un code succinct et lisible. Vous remarquerez que pour utiliser la notation en cascade, vous devez omettre le ';' tant que vous désirez rester sur le même objet.

4.5 Les Constructeurs

4.5.1 Définition d'un constructeur

Pour instancier une classe, nous utilisons le mot-clé `new`, suivi du constructeur correspondant avec des paramètres, si nécessaire. Maintenant, modifions la classe `Personne` et définissons un constructeur avec des paramètres ci-dessous :

The screenshot shows a code editor interface with a dark theme. On the left, there is a snippet of Dart code:

```
class Personne {
  String prenom;
  String nom;
  Personne(String prenom, String nom) {
    this.prenom = prenom;
    this.nom = nom;
  }
  String getNomComplet() => "$prenom $nom";
}

main() {
  // Personne unePersonne = new Personne(); n'est plus supportée
  // car nous devons définir les paramètres du constructeur
  Personne unePersonne = new Personne("Clark", "Kent");
  print(unePersonne.getNomComplet());
}
```

In the top right corner, there is a blue button labeled "RUN". To the right of the code editor, there is a "Console" window showing the output:

```
Console
Clark Kent
```

Le constructeur est également une fonction dans Dart et son rôle est d'initialiser correctement l'instance de la classe. En tant que fonction, elle peut avoir les nombreuses caractéristiques d'une

fonction Dart commune, telles que des arguments - obligatoires ou facultatifs, et nommés ou positionnels. Dans l'exemple précédent, le constructeur a deux arguments obligatoires.

Si vous regardez le corps du constructeur, il utilise le mot-clé `this`. De plus, les noms des paramètres du constructeur sont les mêmes que ceux des champs, ce qui peut provoquer une ambiguïté. Donc, pour éviter cela, nous préfixons les champs d'instance d'objet avec le mot clé `this` lors de l'étape d'attribution de valeur.

Dart fournit une autre façon d'écrire un constructeur comme celui fourni dans l'exemple, en utilisant une syntaxe raccourcie :



```
class Personne {  
  String prenom;  
  String nom;  
  Personne(this.prenom, this.nom);  
  String getNomComplet() => "$prenom $nom";  
}  
  
main() {  
  // Personne unePersonne = new Personne(); n'est plus supportée  
  // car nous devons définir les paramètres du constructeur  
  Personne unePersonne = new Personne("Clark", "Kent");  
  print(unePersonne.getNomComplet());  
}
```

Nous pouvons omettre le corps du constructeur car il ne définit que les valeurs du champ de classe sans aucune configuration supplémentaire qui lui est appliquée.

4.5.2 Les constructeurs nommés

Contrairement à Java et à de nombreux autres langages, Dart n'a pas de surcharge par redéfinition, donc, pour définir des constructeurs alternatifs pour une classe, vous devez utiliser les constructeurs nommés :



```
class Personne {  
  String prenom;  
  String nom;  
  Personne(this.prenom, this.nom);  
  String getNomComplet() => "$prenom $nom";  
  Personne.anonyme(){  
  }  
  Personne.sansNom();  
}
```

Un constructeur nommé est la façon dont vous définissez des constructeurs alternatifs pour une classe. Dans l'exemple précédent, nous avons défini deux constructeurs alternatifs pour une classe `Personne` sans nom.

4.5.3 Le constructeur `factory`

Une autre syntaxe utile dans Dart est le constructeur `factory`, qui permet d'appliquer le modèle, une technique de création qui permet aux classes d'être instanciées sans spécifier le type d'objet résultant.

Supposons que nous ayons les descendants suivants de la classe `Personne`:

```

class Etudiant extends Personne {
    Etudiant(prenom, nom): super(prenom, nom);
}

class Employe extends Personne {
    Employe(prenom, nom): super(prenom, nom);
}

```

Comme vous pouvez le constater, les classes descendantes sont toujours presque les mêmes que la classe **Personne**, car elles n'ajoutent pas encore de fonctionnalités spécifiques. Nous pouvons définir un constructeur **factory** sur la classe **Personne** pour instancier la classe correspondante en fonction de l'argument de type requis :

```

enum TypePersonne {
    etudiant, employe
}

class Etudiant extends Personne {
    Etudiant(prenom, nom): super(prenom, nom);
}

class Employe extends Personne {
    Employe(prenom, nom): super(prenom, nom);
}

class Personne {
    String prenom;
    String nom;
    Personne([this.prenom, this.nom]);

    factory Personne.fromType(prenom, nom,[TypePersonne type]) {
        switch(type) {
            case TypePersonne.etudiant:
                return new Etudiant(prenom, nom);
            case TypePersonne.employe:
                return new Employe(prenom, nom);
        }
        return Personne();
    }
    String getNomCompletn() => "$prenom $nom";
}

main() {
    Personne unePersonne = new Personne.fromType("Clark", "Kent",TypePersonne.employe);
    print(unePersonne.getNomCompletn());
    print(unePersonne.runtimeType);
}

```

Le constructeur est spécifié en ajoutant le mot-clé **factory**, suivi de la définition du constructeur, généralement dans une classe de base ou un type de classe abstraite. Dans notre cas, la classe **Personne** définit un constructeur **factory** basé sur **TypePersonne** spécifié dans l'argument. Si aucun type n'est passé, il crée une classe **Personne** simple en utilisant son constructeur par défaut. Une autre chose importante à noter est que le constructeur **factory** ne remplace pas le constructeur de classe par défaut. Par conséquent, lui et ses descendants peuvent toujours être instanciés directement par l'appelant.

4.5.4 Accesseurs de champs – **get** et **set**

Comme mentionné précédemment, les getters et setters nous permettent d'accéder à un champ sur une classe, et chaque champ a ces accesseurs, même si nous ne les définissons pas.

Dans l'exemple précédent, lorsque nous exécutons `unePersonne.prenom = "Peter"`, nous appelons l'accesseur **set** du champ **prenom** et lui envoyons "Peter" comme paramètre. Également dans l'exemple, l'accesseur **get** est utilisé lorsque nous appelons la méthode `getNomCompletn()`, et il concatène prénom et nom.

Nous pouvons modifier notre classe Personne pour remplacer l'ancienne méthode `getNomComplet()` et l'ajouter en tant que `get`, comme donné dans le code suivant :

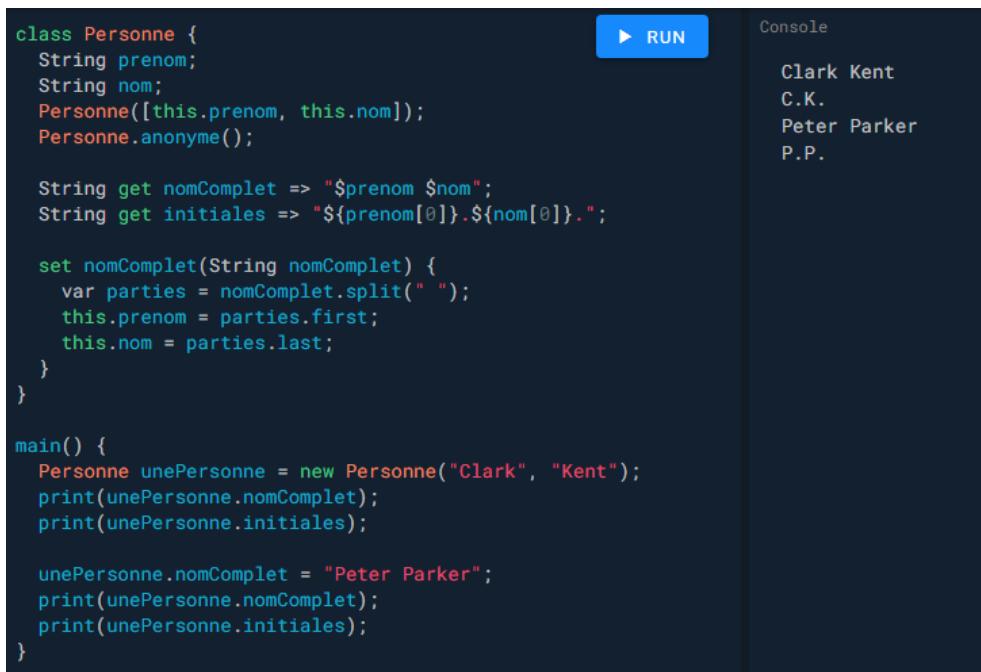


```
class Personne {  
    String prenom;  
    String nom;  
    Personne([this.prenom, this.nom]);  
    Personne.anonyme();  
  
    String get nomCompletn => "$prenom $nom";  
    String get initiales => "${prenom[0]}.${nom[0]}.";  
}  
  
main() {  
    Personne unePersonne = new Personne("Clark", "Kent");  
    print(unePersonne.nomCompletn);  
    print(unePersonne.initiales);  
  
    // unePersonne.nomCompletn = "Peter Parker"; generera une  
    // erreur car la fonction set n'est pas definie pour ce champ  
}
```

Les remarques importantes suivantes peuvent être faites concernant l'exemple précédent :

- Nous n'aurions pas pu définir un `get` ou un `set` avec les mêmes noms de champ : `prenom` et `nom`. Cela nous donnerait une erreur de compilation, car les noms des membres de la classe ne peuvent pas être répétés.
- Le `get initiales` lèverait une erreur pour une personne instanciée par le constructeur nommé anonyme, car il n'aurait pas de valeurs `prenom` et `nom` (équivaut à `null`).
- Nous n'avons pas besoin de toujours définir la paire, `get` et `set`, ensemble. Comme vous pouvez le voir, nous n'avons défini qu'un `get nomCompletn` et non un `set`, nous ne pouvons donc pas modifier `nomCompletn`. (Cela entraîne une erreur de compilation, comme indiqué précédemment.)

Nous aurions pu également écrire un `set` pour `nomCompletn` et définir la logique sous-jacente pour définir `prenom` et `nom`.



```
class Personne {  
    String prenom;  
    String nom;  
    Personne([this.prenom, this.nom]);  
    Personne.anonyme();  
  
    String get nomCompletn => "$prenom $nom";  
    String get initiales => "${prenom[0]}.${nom[0]}.";  
  
    set nomCompletn(String nomCompletn) {  
        var parties = nomCompletn.split(" ");  
        this.prenom = parties.first;  
        this.nom = parties.last;  
    }  
  
    main() {  
        Personne unePersonne = new Personne("Clark", "Kent");  
        print(unePersonne.nomCompletn);  
        print(unePersonne.initiales);  
  
        unePersonne.nomCompletn = "Peter Parker";  
        print(unePersonne.nomCompletn);  
        print(unePersonne.initiales);  
    }  
}
```

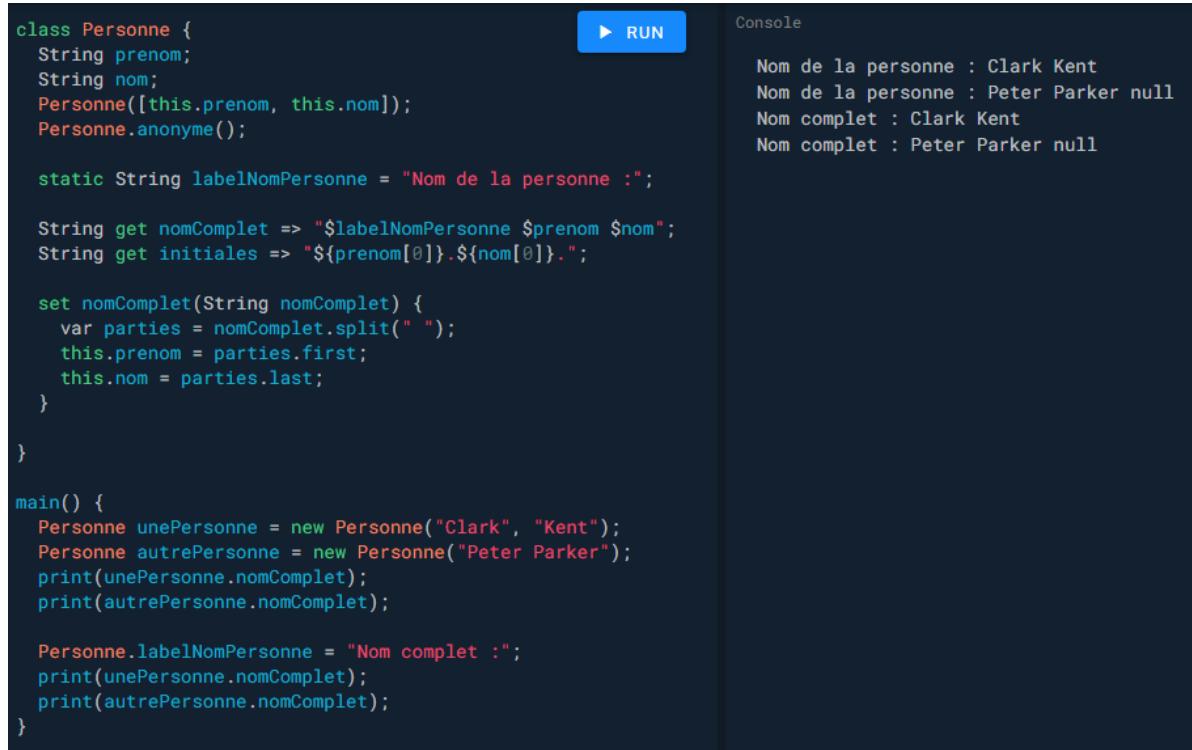
De cette façon, quelqu'un pourrait initialiser le nom d'une personne en définissant `nomComplet` et le résultat serait le même.

4.5.5 Les champs et les méthodes statiques

Comme vous le savez déjà, les champs ne sont rien de plus que des variables qui contiennent des valeurs de l'objet, et les méthodes sont des fonctions simples qui représentent des actions de l'objet. Dans certains cas, vous souhaiterez peut-être partager une valeur ou une méthode entre toutes les instances d'objet d'une classe.

4.5.5.1 *Les champs statiques*

Pour ce cas d'utilisation, vous pouvez leur ajouter le modificateur `static` (voir exemple ci-dessous).



```
class Personne {
    String prenom;
    String nom;
    Personne([this.prenom, this.nom]);
    Personne.anonyme();

    static String labelNomPersonne = "Nom de la personne :";

    String get nomComplet => "$labelNomPersonne $prenom $nom";
    String get initiales => "${prenom[0]}.${nom[0]}.";

    set nomComplet(String nomComplet) {
        var parties = nomComplet.split(" ");
        this.prenom = parties.first;
        this.nom = parties.last;
    }

    main() {
        Personne unePersonne = new Personne("Clark", "Kent");
        Personne autrePersonne = new Personne("Peter Parker");
        print(unePersonne.nomComplet);
        print(autrePersonne.nomComplet);

        Personne.labelNomPersonne = "Nom complet :";
        print(unePersonne.nomComplet);
        print(autrePersonne.nomComplet);
    }
}
```

Console

```
Nom de la personne : Clark Kent
Nom de la personne : Peter Parker null
Nom complet : Clark Kent
Nom complet : Peter Parker null
```

Il est à noter que vous pouvez changer la valeur du champ statique directement sur la classe.

4.5.5.2 *Les méthodes statiques*

Les champs statiques sont associés à la classe, plutôt qu'à toute instance d'objet. Il en va de même pour les définitions de méthodes statiques. Vous pouvez ajouter une méthode statique pour encapsuler l'affichage du nom, comme illustré dans le bloc de code suivant. Dans ce cas, vous pouvez utiliser cette méthode pour afficher une instance de `Personne`, comme nous l'avons fait auparavant :

```

class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  static String labelNomPersonne = "Nom de la personne :";

  String get nomComplet => "$prenom $nom";
  String get initiales => "${prenom[0]}.${nom[0]}.";

  set nomComplet(String nomComplet) {
    var parties = nomComplet.split(" ");
    this.prenom = parties.first;
    this.nom = parties.last;
  }

  static void afficherPersonne(Personne personne) {
    print("$labelNomPersonne ${personne.prenom} ${personne.nom}");
  }
}

main() {
  Personne unePersonne = new Personne("Clark", "Kent");
  Personne autrePersonne = new Personne("Peter", "Parker");

  Personne.afficherPersonne(unePersonne);
  Personne.afficherPersonne(autrePersonne);
}

```

▶ RUN

Console

Nom de la personne : Clark Kent
Nom de la personne : Peter Parker

Nous pourrions modifier le `get nomComplet` sur la classe `Personne` pour ne pas utiliser le champ statique `labelNomPersonne`, pour avoir plus de sens et obtenir des résultats distincts selon nos besoins :

```

class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  static String labelNomPersonne = "Nom de la personne :";

  String get nomComplet => "$prenom $nom";
  String get initiales => "${prenom[0]}.${nom[0]}.";

  static void afficherPersonne(Personne personne) {
    print("$labelNomPersonne ${personne.prenom} ${personne.nom}");
  }
}

main() {
  Personne unePersonne = new Personne("Clark", "Kent");
  Personne autrePersonne = new Personne("Peter", "Parker");

  print(unePersonne.nomComplet);
  print(autrePersonne.nomComplet);

  Personne.afficherPersonne(unePersonne);
  Personne.afficherPersonne(autrePersonne);
}

```

▶ RUN

Console

Clark Kent
Peter Parker
Nom de la personne : Clark Kent
Nom de la personne : Peter Parker

Comme vous pouvez le voir, les champs et méthodes statiques nous permettent d'ajouter des comportements spécifiques aux classes en général.

4.5.6 Héritage de classe

4.5.6.1 Principe

En plus de l'héritage implicite du type `Object`, Dart nous permet d'étendre des classes définies en utilisant le mot-clé `extends`, où tous les membres de la classe parente sont hérités, à l'exception des

constructeurs. Voyons maintenant l'exemple suivant, où nous créons une classe enfant pour la classe **Personne** existante :

The screenshot shows a Dart code editor with two tabs. The left tab contains the code for the **Etudiant** class and the **Personne** class. The right tab shows the output in the "Console" window.

```
class Etudiant extends Personne {
  String surnom;
  Etudiant(String prenom, String nom, this.surnom):super(prenom, nom);

  @override
  String toString() => "$nomComplet, connu comme $surnom";
}

class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  static String labelNomPersonne = "Nom de la personne :";

  String get nomComplet => "$prenom $nom";
  String get initiales => "${prenom[0]}.${nom[0]}.";

  static void afficherPersonne(Personne personne) {
    print("$labelNomPersonne ${personne.prenom} ${personne.nom}");
  }
}

main() {
  Etudiant etudiant = new Etudiant("Clark", "Kent", "Kal-El");
  print(etudiant); // équivalent à appeler etudiant.toString()
}
```

Console
Clark Kent, connu comme Kal-El

Les observations suivantes peuvent être faites concernant l'exemple précédent :

- **Etudiant** : La classe **Etudiant** définit son propre constructeur. Cependant, il appelle le constructeur de la classe **Personne**, en passant les paramètres requis. Ceci est fait avec le mot clé **super**.
- **@override** : Il existe une méthode **toString()** remplacée sur la classe **Etudiant**. C'est là que l'héritage prend tout son sens - nous modifions le comportement d'une classe parent (**Object**, dans ce cas) sur la classe enfant.
- **print(etudiant)** : Comme vous pouvez le voir dans la déclaration **print(etudiant)**, nous n'appelons aucune méthode; la méthode **toString()** est appelée pour nous implicitement.

4.5.6.2 La méthode **toString()**

La méthode **toString()** est un excellent exemple courant de remplacement du comportement des parents. L'objectif de cette méthode est de renvoyer une représentation **String** de l'objet :

The screenshot shows a simplified version of the **Etudiant** class with a single **toString()** method.

```
class Etudiant extends Personne {
  String surnom;
  Etudiant(String prenom, String nom, this.surnom):super(prenom, nom);

  @override
  String toString() => "$nomComplet, connu comme $surnom";
}

main() {
  Etudiant etudiant = new Etudiant("Clark", "Kent", "Kal-El");
  print(etudiant); // équivalent à appeler etudiant.toString()
}
```

Comme vous pouvez le voir, cela rend le code plus propre et nous fournissons une bonne représentation textuelle de l'objet qui peut aider à comprendre les enregistrements, la mise en forme du texte, etc.

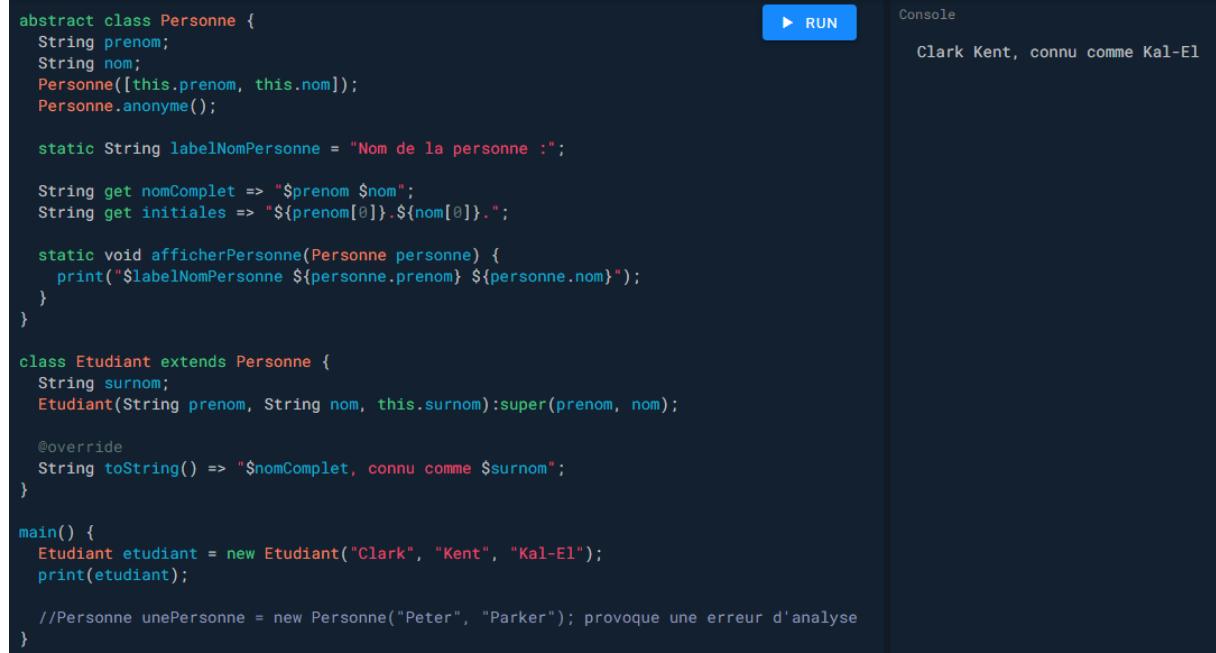
4.5.7 Interfaces, classes abstraites et mixins

Dans Dart, les classes abstraites et les interfaces sont étroitement liées les unes aux autres. Cela est dû au fait que Dart implémente les interfaces d'une manière subtilement différente de la plupart des langages classiques. Jetons un coup d'œil aux classes abstraites avant de les lier au sujet des interfaces implicites.

4.5.7.1 Les classes abstraites

En POO, les classes abstraites sont des classes qui ne peuvent pas être instanciées, ce qui a beaucoup de sens, selon le contexte et le niveau d'abstraction dans un programme.

Par exemple, notre classe `Personne` pourrait être abstraite si nous voulons nous assurer qu'elle n'existe que dans le contexte du programme s'il s'agit d'une instance `Etudiant` ou d'une autre classe enfant :



```
abstract class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  static String labelNomPersonne = "Nom de la personne :";

  String get nomComplet => "$prenom $nom";
  String get initiales => "${prenom[0]}.${nom[0]}.";

  static void afficherPersonne(Personne personne) {
    print("$labelNomPersonne ${personne.prenom} ${personne.nom}");
  }
}

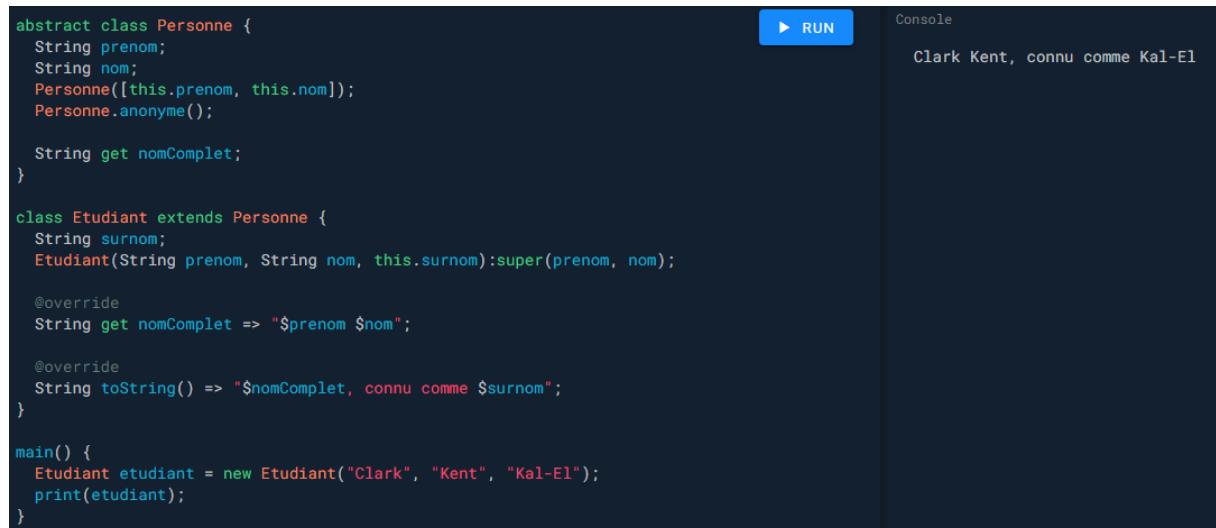
class Etudiant extends Personne {
  String surnom;
  Etudiant(String prenom, String nom, this.surnom):super(prenom, nom);

  @override
  String toString() => "$nomComplet, connu comme $surnom";
}

main() {
  Etudiant etudiant = new Etudiant("Clark", "Kent", "Kal-El");
  print(etudiant);

  //Personne unePersonne = new Personne("Peter", "Parker"); provoque une erreur d'analyse
}
```

Comme vous pouvez le voir, nous ne pouvons plus instancier une classe `Personne`, juste une classe enfant, `Etudiant`. Une classe abstraite peut avoir des membres abstraits sans implémentation, ce qui lui permet d'être implémentée par les types enfants qui les étendent :



```
abstract class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  String get nomComplet;
}

class Etudiant extends Personne {
  String surnom;
  Etudiant(String prenom, String nom, this.surnom):super(prenom, nom);

  @override
  String get nomComplet => "$prenom $nom";

  @override
  String toString() => "$nomComplet, connu comme $surnom";
}

main() {
  Etudiant etudiant = new Etudiant("Clark", "Kent", "Kal-El");
  print(etudiant);
}
```

Le get nomComplet de la classe Personne précédente est maintenant abstrait, car il n'a pas d'implémentation. Il est de la responsabilité de l'enfant d'implémenter ce membre. La classe Etudiant implémente le get nomComplet car, si ce n'était pas le cas, vous auriez une erreur de compilation.

4.5.7.2 Les Interfaces

Dart n'a pas le mot-clé interface mais nous permet d'utiliser les interfaces d'une manière subtilement différente de ce à quoi vous êtes habitué. Toutes les déclarations de classe sont elles-mêmes des interfaces. Cela signifie que, lorsque vous définissez une classe dans Dart, vous définissez également une interface qui peut être implémentée et pas seulement étendue par d'autres classes. C'est ce qu'on appelle les interfaces implicites dans le monde de Dart.

Sur cette base, notre précédente classe Personne est également une interface Personne qui pourrait être implémentée, au lieu d'être étendue, par la classe Etudiant :

```
class Etudiant implements Personne {
    String surnom;
    @override
    String prenom;
    @override
    String nom;
    Etudiant(String prenom, String nom, this.surnom);
    @override
    String get nomComplet => "$prenom $nom";
    @override
    String toString() => "$nomComplet, connu comme $surnom";
}
```

Notez que, en général, le code ne change pas trop, sauf dans la mesure où les membres sont désormais définis dans la classe Etudiant. La classe Personne n'est qu'un contrat que la classe Etudiant doit adopter et implémenter.

4.5.7.3 Les mixins - ajout d'un comportement à une classe

En POO, les mixins sont un moyen d'inclure des fonctionnalités sur une classe sans avoir besoin d'associations entre les parties, comme l'héritage.

Les contextes les plus courants dans lesquels les mixins peuvent être utilisés sont dans les endroits où plusieurs héritages peuvent être nécessaires, car c'est un moyen facile pour les classes d'utiliser des fonctionnalités communes.

Dans Dart, il existe plusieurs façons de déclarer un mixin:

- ⊕ En déclarant une classe et en l'utilisant comme mixin, en lui permettant d'être également utilisé comme objet
- ⊕ En déclarant une classe abstraite, en lui permettant d'être utilisée comme mixin ou être hérité, mais pas instancié
- ⊕ En le déclarant comme mixin, lui permettant d'être utilisé uniquement comme mixin

Voyons maintenant un exemple de déclaration d'une fonctionnalité que notre précédente classe Personne pourrait avoir. Par exemple, pensons aux professions qu'une personne pourrait avoir - certaines personnes peuvent avoir des compétences spécifiques et des compétences communes.

Les mixins peuvent être idéaux pour ce cas d'utilisation car nous pouvons ajouter les compétences à une profession sans avoir besoin d'étendre une classe commune plus générique ou d'implémenter une interface dans chacune d'elle. Comme l'implémentation serait probablement la même, cela provoquerait des duplications de code :

```
class CompetencesProgrammation {  
    code() {  
        print("Ecrit du code...");  
    }  
}  
  
class CompetencesGestion {  
    gere() {  
        print("Gère du projet...");  
    }  
}
```

Dans l'exemple précédent, nous avons créé deux classes de compétences professionnelles, `CompetencesProgrammation` et `CompetencesGestion`. Maintenant, nous pouvons les utiliser en ajoutant le mot-clé `with` à la définition de classe, par exemple :

```
class DeveloppeurSenior extends Personne with CompetencesProgrammation, CompetencesGestion {  
    DeveloppeurSenior(String prenom, String nom) : super(prenom, nom);  
}  
  
class DeveloppeurJunior extends Personne with CompetencesProgrammation {  
    DeveloppeurJunior(String prenom, String nom) : super(prenom, nom);  
}
```

Les deux classes auront la méthode `code()` sans avoir besoin de l'implémenter dans chaque classe, car elle est déjà implémentée dans le mixin `CompetencesProgrammation`.

Comme mentionné précédemment, il existe plusieurs façons de déclarer un mixin. Dans l'exemple précédent, nous avons utilisé une définition de classe simple. De cette façon, la classe `CompetencesProgrammation` peut être étendue comme une classe normale ou même implémentée en tant qu'interface (en perdant la propriété mixin) :

```
class CompetencesProgrammationAvancee extends CompetencesProgrammation {  
    faitLeCafe() {  
        print("Fait le café...");  
    }  
}
```

Une autre façon d'écrire un mixin est d'utiliser le mot-clé `mixin`:

```
mixin CompetencesProgrammation {  
    code() {  
        print("Ecrit du code...");  
    }  
}  
  
mixin CompetencesGestion {  
    gere() {  
        print("Gère du projet...");  
    }  
}
```

L'écriture de mixins de cette manière empêche les comportements indésirables car les mixins ne peuvent pas être étendus et sont destinés à être utilisés correctement. Les classes qui utilisent des mixins restent les mêmes. Une autre chose que nous pouvons faire est de limiter les classes qui peuvent utiliser un certain mixin. Pour ce faire, nous devons spécifier la superclasse requise en utilisant le mot-clé `on` :

The screenshot shows a Dart code editor interface. On the left, the code is written in Dart. On the right, there is a 'RUN' button, a 'Console' output area, and a 'Documentation' link.

```
mixin CompetencesProgrammation on Personne {
  code() {
    print("Ecrit du code...");
  }
}

mixin CompetencesGestion on Personne {
  gere() {
    print("Gère du projet...");
  }
}

class DeveloppeurSenior extends Personne with CompetencesProgrammation, CompetencesGestion {
  DeveloppeurSenior(String prenom, String nom) : super(prenom, nom);

  @override
  String get nomComplet => "$prenom $nom";
  @override
  String toString() => "$nomComplet, Développeur Senior";
}

class DeveloppeurJunior extends Personne with CompetencesProgrammation {
  DeveloppeurJunior(String prenom, String nom) : super(prenom, nom);

  @override
  String get nomComplet => "$prenom $nom";
  @override
  String toString() => "$nomComplet, Développeur Junior";
}

abstract class Personne {
  String prenom;
  String nom;
  Personne([this.prenom, this.nom]);
  Personne.anonyme();

  String get nomComplet;
}

main() {
  DeveloppeurSenior senior = new DeveloppeurSenior("Clark", "Kent");
  DeveloppeurJunior junior = new DeveloppeurJunior("Peter", "Parker");
  print(senior);
  print(junior);
}
```

Console output:

```
Clark Kent, Développeur Senior
Peter Parker, Développeur Junior
```

Documentation link:

[Documentation](#)

4.5.8 Les classes appelables, les fonctions et variables de niveau supérieur

Dart est très flexible en permettant au développeur de prendre le contrôle de tous les morceaux de son code et, contrairement à de nombreux langages, il n'y a pas un moyen unique de faire quelque chose.

Comme Dart propose de combiner les avantages des concepts de la POO modernes avec les concepts traditionnels, vous pouvez toujours choisir quand et où appliquer les différentes approches.

4.5.8.1 *Les classes appelables*

De la même manière que les fonctions Dart ne sont rien de plus que des objets, les classes Dart peuvent également se comporter comme des fonctions, c'est-à-dire qu'elles peuvent être appelées, prendre des arguments et renvoyer quelque chose en conséquence.

La syntaxe pour émuler une fonction dans une classe est la suivante :

```
class DoitEcrireUnProgramme { // c'est une classe simple
  String langage;
  String plateforme;

  DoitEcrireUnProgramme (this.langage, this.plateforme);

  // cette méthode spéciale nommée 'call' fait que la classe se comporte comme une fonction
  bool call (String categorie) {
    if (langage == "Dart" && plateforme == "Flutter") {
      return categorie != "à faire";
    }
    return false;
  }

  main() {
    var doitEcrire = DoitEcrireUnProgramme("Dart", "Flutter");

    print(doitEcrire("à faire")); // affiche false
  }
}
```

▶ RUN

Console
false

Comme vous pouvez le voir, la variable `doitEcrire` est un objet, une instance de la classe `DoitEcrireUnProgramme`, mais peut également être appelée comme une fonction normale en passant un paramètre et en utilisant sa valeur de retour. Cela est possible en raison de l'existence de la méthode `call()` définie dans la classe. La méthode `call()` est une méthode spéciale de Dart. Chaque classe qui la définit peut se comporter comme une fonction Dart normale.

4.5.8.2 Les fonctions et variables de premier niveau

Nous avons vu que les fonctions et les variables de Dart peuvent être liées à des classes en tant que membres - champs et méthodes.

La manière d'écrire des fonctions de premier niveau vous est également déjà connue. Dans l'introduction à Dart, nous avons écrit la fonction Dart la plus célèbre : le point d'entrée de chaque application, `main()`.

Pour les variables, la manière de déclarer est la même. Nous le laissons simplement hors de toutes fonctions, de sorte qu'elles soient accessible globalement sur l'application :

```
var nombreGlobal = 100;
final nombreGlobalFinal = 1000;

void afficherBonjour() {
  print("""De portée globale pour Dart.
  Il s'agit d'un nombre de premier niveau: $nombreGlobal
  Il s'agit d'un nombre final de premier niveau: $nombreGlobalFinal
  """);
}

main() {
  // la fonction de niveau supérieur Dart la plus célèbre
  afficherBonjour(); // affiche la valeur par défaut
  nombreGlobal = 0;
  // nombreGlobalFinal = 0; // ne compile pas car il s'agit d'une variable finale
  afficherBonjour(); // affiche la nouvelle valeur
}
```

▶ RUN

Console

De portée globale pour Dart.
Il s'agit d'un nombre de premier niveau: 100
Il s'agit d'un nombre final de premier niveau: 1000

De portée globale pour Dart.
Il s'agit d'un nombre de premier niveau: 0
Il s'agit d'un nombre final de premier niveau: 1000

4.6 Comprendre les bibliothèques et les packages Dart

Les bibliothèques sont un moyen de structurer un projet basé sur la modularité, ce qui vous permet de diviser le code sur plusieurs fichiers et de partager un morceau de code ou un module avec d'autres développeurs.

De nombreux langages de programmation utilisent des bibliothèques pour fournir cette modularité au développeur, et Dart n'est pas différent. Dans Dart, ces bibliothèques ont également un autre rôle important en plus de la structuration du code. Elles déterminent ce qui est visible ou non pour les autres bibliothèques ou programme Dart.

Avant d'entrer dans les packages Dart, vous devez comprendre la plus petite unité qui les compose : la bibliothèque. Tout d'abord, explorons comment utiliser une bibliothèque dans notre package et, par la suite, apprenons à définir une bibliothèque dans Dart.



NB : A partir de là, nous devons quitter « DartPad » pour travailler sur une machine virtuelle Dart. Pour cela, je vous propose d'utiliser Visual Studio Code pour la création, l'édition et l'exécution de vos programmes Dart (voir annexe 4 :procédure d'utilisation d'une VM Dart).

4.6.1 Importer et utiliser une bibliothèque

Dans l'introduction à Dart, dans la section « Fonctions », nous avons importé une bibliothèque pour utiliser l'annotation `@required` sur certains paramètres. Maintenant, explorons l'instruction d'importation plus en détail.

Pour définir une bibliothèque, nous créons simplement un fichier Dart contenant du code. Dans cet exemple, nous avons défini une bibliothèque simple avec les classes `Personne`, `Etudiant` et `Employe` à côté de l'énumération `TypePersonne` :

```
// Librairie : lib_personne.dart
enum TypePersonne { etudiant, employe }

class Personne {
    String prenom;
    String nom;
    TypePersonne type;

    Personne([this.prenom, this.nom]);
    String get nomComplet => "$prenom $nom";
}

class Etudiant extends Personne {
    Etudiant([prenom, nom]) : super(prenom, nom) {
        type = TypePersonne.etudiant;
    }
}

class Employe extends Personne {
    Employe([prenom, nom]) : super(prenom, nom) {
        type = TypePersonne.employe;
    }
}
```

Pour l'importer, nous pouvons simplement ajouter `import 'library_path'`; instruction au début du fichier et avant tout code:

The screenshot shows a Visual Studio Code interface with the following details:

- Explorateur:** Shows the project structure under "APP46".
- ÉDITEURS:** Displays two files: "app46.dart" and "lib_personne.dart".
- Terminal:** Shows the output of a run command:

```
Personne : Clark Kent, type: null
Etudiant : Peter Parker, type: TypePersonne.etudiant
Exited
```
- Barre d'état:** Indique la ligne de code courante (L 10, col 1), les espaces (Espaces : 2), le codage (UTF-8 CRLF), Dart, et Dart from Flutter: 1.20.2.

Comme les fichiers sont dans le même répertoire, le chemin d'importation est simplement le nom du fichier. Après avoir ajouté l'instruction import, nous pouvons utiliser n'importe quel code disponible, de la même manière que nous l'avons fait avec les classes `Personne` et `Etudiant`.

4.6.2 Renommer ou masquer des éléments de la bibliothèque

Si vous regardez l'exemple précédent, vous remarquerez que nous n'avons pas utilisé toutes les classes disponibles de la bibliothèque `lib_personne`. Pour rendre le code plus propre et moins sensible aux erreurs et aux conflits de noms, nous pouvons utiliser le mot-clé `show`, qui nous permet d'importer uniquement les identifiants que nous voulons utiliser efficacement dans notre code :

```
bin > app46.dart > ...
1   import 'lib_personne.dart' show Personne, Etudiant;
```

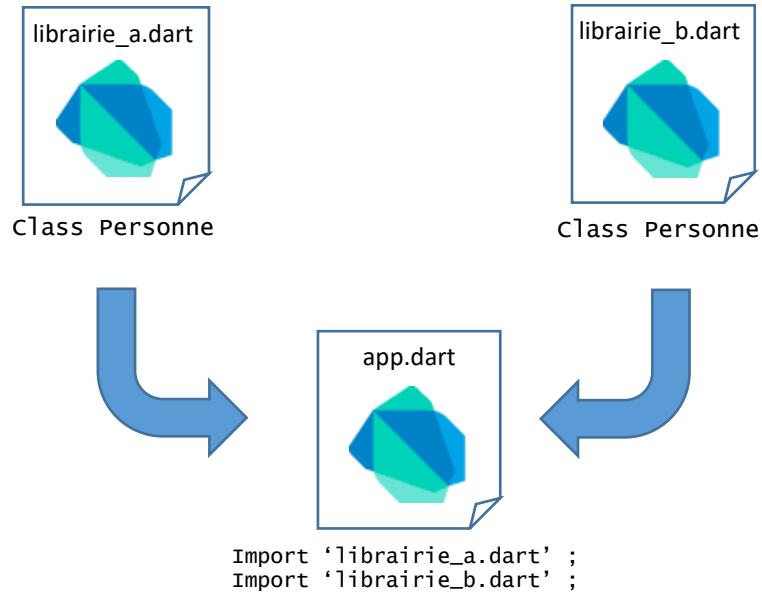
Nous pouvons également spécifier les identifiants que nous ne souhaitons pas importer explicitement en utilisant le mot-clé `hide`. Dans ce cas, nous importerons tous les identifiants de la bibliothèque à l'exception de ceux situés après le mot-clé `hide`:

```
bin > app46.dart > ...
1   import 'lib_personne.dart' hide Employe;
```

4.6.3 Importer des préfixes dans des bibliothèques

Dans Dart, il n'y a pas de définition d'espace de noms ou quelque chose qui identifie de manière unique une bibliothèque dans le contexte dans lequel elle est utilisée, de sorte que des conflits peuvent survenir lors de la création de noms d'identificateurs ; autrement dit, les bibliothèques peuvent définir une fonction de niveau supérieur ou même une classe du même nom. Bien que nous puissions utiliser les modificateurs `show` et `hide` pour définir explicitement les membres que nous voulons importer à

partir d'une bibliothèque, cela n'est pas suffisant pour résoudre le problème car, parfois, nous pouvons être intéressés par une classe ou une fonction de niveau supérieur portant le même nom dans différentes bibliothèques :



Quelle classe Personne utiliser ?

Heureusement, Dart a un moyen de contourner ce problème. Le mot-clé `as` peut être ajouté après une instruction d'importation pour définir un préfixe pour tous les identificateurs de la bibliothèque importée :

```
bin > app46.dart > ...
1 import 'lib_personne_a.dart' as librairieA;
2 import 'lib_personne_b.dart' as librairieB;
3
4 Run | Debug
5 void main() {
6     librairieA.Personne unePersonne = librairieA.Personne("Clark", "Kent");
7     librairieB.Personne unEtudiant = librairieB.Etudiant("Peter", "Parker");
8
9     print("Personne : ${unePersonne.nomComplet}, type: ${unePersonne.type}");
10    print("Etudiant : ${unEtudiant.nomComplet}, type: ${unEtudiant.type}");
11 }
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL

```
Personne : LIB_A : Clark Kent, type: null
Etudiant : LIB_B : Peter Parker, type: TypePersonne.etudiant
Exited
```

Comme vous pouvez le voir, sans ce préfixe, nous n'avons pas de moyen d'identifier la classe **Personne** à utiliser. Il en va de même pour tout identifiant de bibliothèque publique, tel qu'une fonction ou une variable. Après avoir spécifié le préfixe, nous devons l'ajouter à chaque appel à un membre de cette bibliothèque, pas seulement ceux qui sont en conflit.

4.6.4 Définition des répertoires d'importation

Dans les exemples précédents, nous avons importé une bibliothèque de fichiers locale qui se trouve dans le même répertoire que le client de la bibliothèque, nous avons donc juste spécifié le nom de fichier.

Cependant, ce n'est pas le cas lorsque vous utilisez des packages Dart de tiers. Dans ce cas, les fichiers n'existeront pas forcément dans le même répertoire, alors voyons comment nous pouvons importer une bibliothèque Dart extraite d'un package externe.

Il existe plusieurs façons de spécifier les chemins de bibliothèque dans l'instruction d'importation, et nous en avons déjà utilisé deux : l'importation relative de fichiers et l'importation à partir d'un package. Maintenant, jetons un coup d'œil à chacun d'eux plus en détail. Supposons que nous ayons un répertoire de package « `mon_package` » contenant deux fichiers : `a.dart` et `b.dart`. Pour les importer, nous pouvons utiliser plusieurs approches :

- Un chemin de fichier relatif : Ceci est similaire à la méthode que nous avons utilisée dans l'exemple précédent, car les bibliothèques étaient dans le même dossier. Nous pouvons simplement mettre le chemin relatif vers le fichier de bibliothèque que nous voulons importer, comme suit :

```
import 'mon_package/a.dart';
import 'mon_package/b.dart';
```

- Un chemin de fichier absolu : Nous pouvons ajouter le chemin absolu sur l'ordinateur à un fichier de bibliothèque en ajoutant le préfixe `file://URI` du chemin d'importation :

```
import "file:///c:/tools/mon_package/a.dart";
import "file:///c:/tools/mon_package/b.dart";
```

- Une URL sur le Web : De la même manière que l'utilisation d'un chemin de fichier absolu, nous pouvons ajouter l'URL d'un site Web contenant le code source d'une bibliothèque directement sur le protocole `http://` :

```
import "http://dartpackage.com/dart_package/mon_package/a.dart";
```

- Un package: c'est la manière la plus courante d'importer une bibliothèque. Ici, nous spécifions le chemin de la bibliothèque à partir de la racine du package. Dans le cas de l'importation d'une bibliothèque locale, elle va de la racine du paquet, dans l'arborescence des fichiers sources jusqu'au fichier de la bibliothèque :

```
import 'package:mon_package/a.dart';
import 'package:mon_package/b.dart';
```

La méthode de package est la méthode recommandée pour importer des bibliothèques, car elle fonctionne bien avec les bibliothèques locales (c'est-à-dire les fichiers et bibliothèques locaux de votre projet) et est la manière d'utiliser les bibliothèques fournies à partir de packages tiers.

4.6.5 Création de bibliothèques Dart

Une bibliothèque Dart peut être composée d'un seul fichier ou de plusieurs fichiers. De la manière la plus courante et recommandée, lorsque vous créez un fichier, vous créez une petite bibliothèque.

Mais, si vous préférez, vous pouvez diviser une définition de bibliothèque en plusieurs fichiers. Bien que moins courant, elle peut être utile en fonction du contexte, notamment lorsque vous travaillez avec des classes très interdépendantes, par exemple.

La décision de fractionner est importante, non seulement pour l'encapsulation, mais également pour la manière dont les clients de la bibliothèque vont les importer et les utiliser.

Disons, par exemple, que nous avons deux classes étroitement couplées qui doivent vivre ensemble pour qu'elles fonctionnent. Les diviser en différentes bibliothèques obligera les clients à importer les deux bibliothèques. Ce n'est pas le moyen le plus pratique, il est donc très important de faire attention au fractionnement de la bibliothèque lors de la création de bibliothèques.

Avant d'entrer dans d'autres façons de définir une bibliothèque, nous devons examiner la confidentialité de la bibliothèque ; cela aide à l'encapsulation, ce qui permet de comprendre plus facilement pourquoi nous devons diviser correctement une bibliothèque en plusieurs fichiers ou non.

4.6.5.1 Confidentialité des membres de la bibliothèque

Le moyen le plus courant de contrôler la confidentialité (encapsulation de code), dans la plupart des langages, se produit au niveau de la classe. C'est en ajoutant un mot-clé spécifique qui identifie le niveau d'accès du membre, tel que `protected` ou `private` dans le langage Java.

Dans Dart, chaque identificateur, par défaut, est accessible de n'importe quel endroit, à l'intérieur et à l'extérieur de la bibliothèque, sauf s'il est précédé d'un caractère `_` (underscore). Cela signifie qu'il devient privé pour les bibliothèques déclarantes, l'empêchant d'être accessible de l'extérieur. Jetez un œil à l'exemple suivant, où nous avons utilisé le préfixe `_`.

Le méta-package Dart fournit l'annotation `@protected`. Lorsqu'il est ajouté à un membre de classe, il indique que le membre doit être utilisé uniquement à l'intérieur de la classe ou de ses sous-types.

De plus, notez que cette partie de Dart est très susceptible de changer dans les versions futures, car une partie de la communauté Dart a été influencée par Java et d'autres langages orientés objet, où le contrôle de la confidentialité a lieu au niveau de la classe.

4.6.5.2 La définition d'une bibliothèque

Dart a un mot-clé pour définir une bibliothèque - `library`, comme vous pouvez vous y attendre. Bien que facultatif, ce mot-clé est très utile lors de la création de plusieurs bibliothèques de fichiers ou pour créer de la documentation pour les bibliothèques avant de les publier en tant qu'API.

NB : Dart a l'outil `dartdoc` pour générer de la documentation HTML pour les packages Dart. Pour utiliser cet outil, nous devons rédiger des commentaires d'une manière spécifique, et nous l'explorerons plus en détail dans la suite de ce cours.

Voyons comment définir une bibliothèque à l'aide de ce mot-clé, et les multiples approches qui peuvent être adoptées lors de la création de bibliothèques pour faire l'encapsulation correcte et rendre l'utilisation de la bibliothèque plus concise.

4.6.5.2.1 Une bibliothèque sur un fichier unique

La manière la plus simple de définir une bibliothèque consiste à ajouter tout le code interdépendant, c'est-à-dire les classes, les fonctions de niveau supérieur et les variables dans un seul fichier. Par exemple, notre bibliothèque **Personne** précédente est la suivante :

```
1 // Librairie : lib_personne.dart
2
3 enum TypePersonne { etudiant, employe }
4
5 class Personne {
6   String prenom;
7   String nom;
8   TypePersonne _type;
9
10 Personne({this.prenom, this.nom});
11 String get nomComplet => "${_type}|: $prenom $nom";
12 }
13
14 class Etudiant extends Personne {
15   Etudiant({prenom, nom}) : super(prenom: prenom, nom: nom) {
16     _type = TypePersonne.etudiant;
17   }
18 }
19
20 class Employe extends Personne {
21   Employe([prenom, nom]) : super(prenom: prenom, nom: nom) {
22     _type = TypePersonne.employe;
23   }
24 }
```

Il n'y a rien de nouveau à noter ici dans la définition du fichier, juste les deux observations suivantes :

- Le fichier, en lui-même, est une bibliothèque, nous n'avons donc pas besoin de déclarer quoi que ce soit explicitement.
- Le champ `_type` est privé pour la bibliothèque, c'est-à-dire qu'il n'est accessible que par le code de cette même bibliothèque.

Disons que nous essayons d'utiliser les classes de cette bibliothèque, comme suit:

The screenshot shows the Visual Studio Code interface with the following details:

- Explorateur (Left Panel):** Shows the project structure with files like `lib_personne.dart`, `app46.dart`, and `CHANGELOG.md`.
- app46.dart File (Center):** Contains Dart code that imports `lib_personne.dart` and defines a `main` function. Inside `main`, it creates instances of `Personne` and `Etudiant` and prints their names and types to the console.
- Console (Bottom):** Displays the output of the `print` statements:

```
Personne : null : Clark Kent
Etudiant : TypePersonne.etudiant : Peter Parker
Exited
```
- Status Bar:** Shows file information like "L 12, col 44 Espaces : 2 UTF-8 CRLF Dart Dart from Flutter: 1.20.2".

Comme vous pouvez le voir, nous avons accès à tous les identifiants publics de la bibliothèque précédemment définie. Nous ne pouvons pas accéder à la propriété `_type` pour définir la valeur, bien que, dans la méthode `toString()` de la classe `Personne`, sa valeur soit accessible. Bien qu'il soit tentant de définir tout le code associé dans un seul fichier, cela peut devenir plus difficile à maintenir, car le code et sa complexité augmentent avec le temps. Utilisez plutôt ceci pour des types de définitions simples qui sont peu susceptibles de changer avec le temps.

4.6.5.2.2 Diviser les bibliothèques en plusieurs fichiers

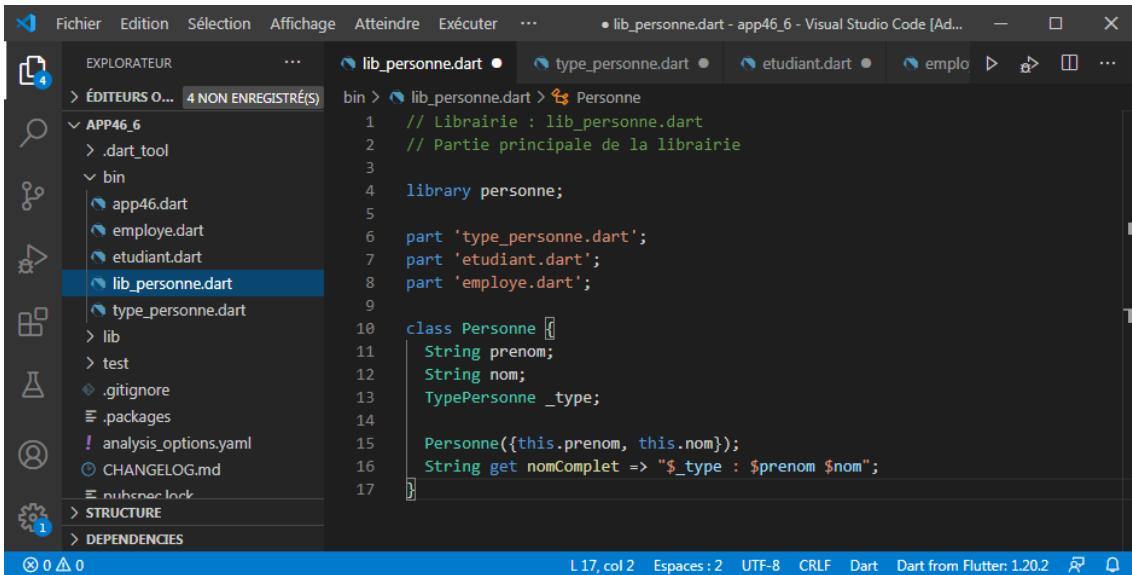
Nous avons vu l'approche du fichier unique pour définir une bibliothèque, alors explorons maintenant comment diviser la définition d'une bibliothèque en plusieurs fichiers pour nous permettre d'organiser le projet en petits morceaux réutilisables (ce qui est le véritable objectif des bibliothèques). Pour définir une bibliothèque à plusieurs fichiers, nous pouvons utiliser les instructions combinées `part`, `part of` et `library`:

`part` : Cela permet à une bibliothèque de spécifier qu'elle est composée de petites parties de bibliothèque.

`part of` : La petite partie bibliothèque spécifie la bibliothèque à laquelle elle est associée.

`library` : Ceci permet d'utiliser les instructions précédentes, car nous devons relier les fichiers de la partie principale de la bibliothèque.

Examinons à quoi ressemble l'exemple précédent en utilisant les instructions `part`:



```
// Librairie : lib_personne.dart
// Partie principale de la librairie
library personne;

part 'type_personne.dart';
part 'etudiant.dart';
part 'employe.dart';

class Personne {
  String prenom;
  String nom;
  TypePersonne _type;

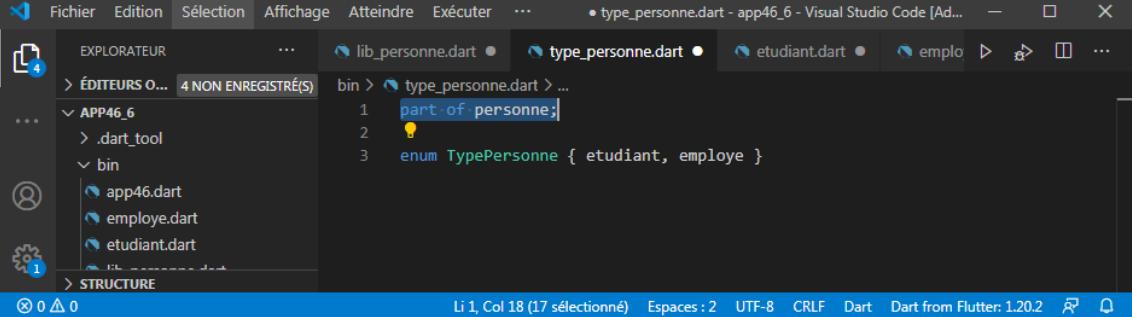
  Personne({this.prenom, this.nom});
  String get nomComplet => "${_type} : $prenom $nom";
}
```

Faisons quelques observations sur le code précédent, comme suit :

- Le mot-clé `library` est suivi de l'identifiant de la bibliothèque, `personne`, dans ce cas. Il est recommandé de nommer l'identificateur en utilisant uniquement des caractères minuscules et le caractère de soulignement comme séparateur. Notre exemple pourrait s'appeler n'importe quoi comme `lib_personne` ou `librairie_personne`.
- Les composants de la bibliothèque sont répertoriés juste en dessous de la définition de la bibliothèque.
- Le code, en lui-même, ne change pas.

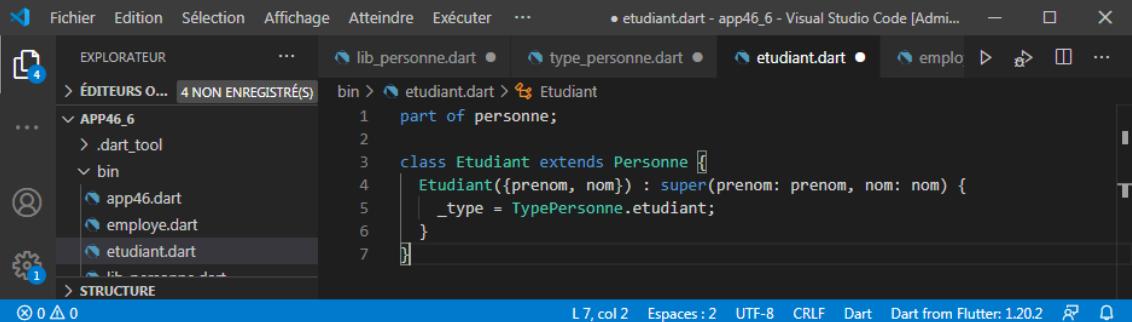
La syntaxe de la part est définie comme suit :

La partie TypePersonne est définie dans le fichier « *type_personne.dart* »:



```
part of personne;
```

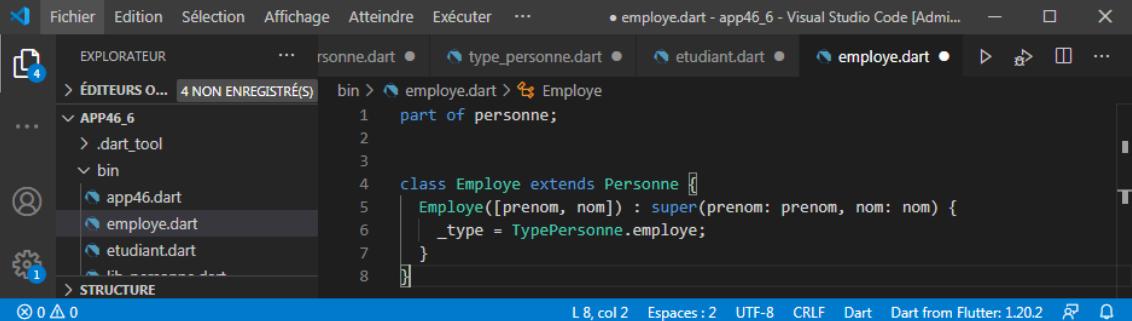
La partie Etudiant est définie dans le fichier « *etudiant.dart* » :



```
part of personne;

class Etudiant extends Personne {
    Etudiant([prenom, nom]) : super(prenom, nom) {
        _type = TypePersonne.etudiant;
}
```

La partie Employe est définie dans le fichier « *employe.dart* »:

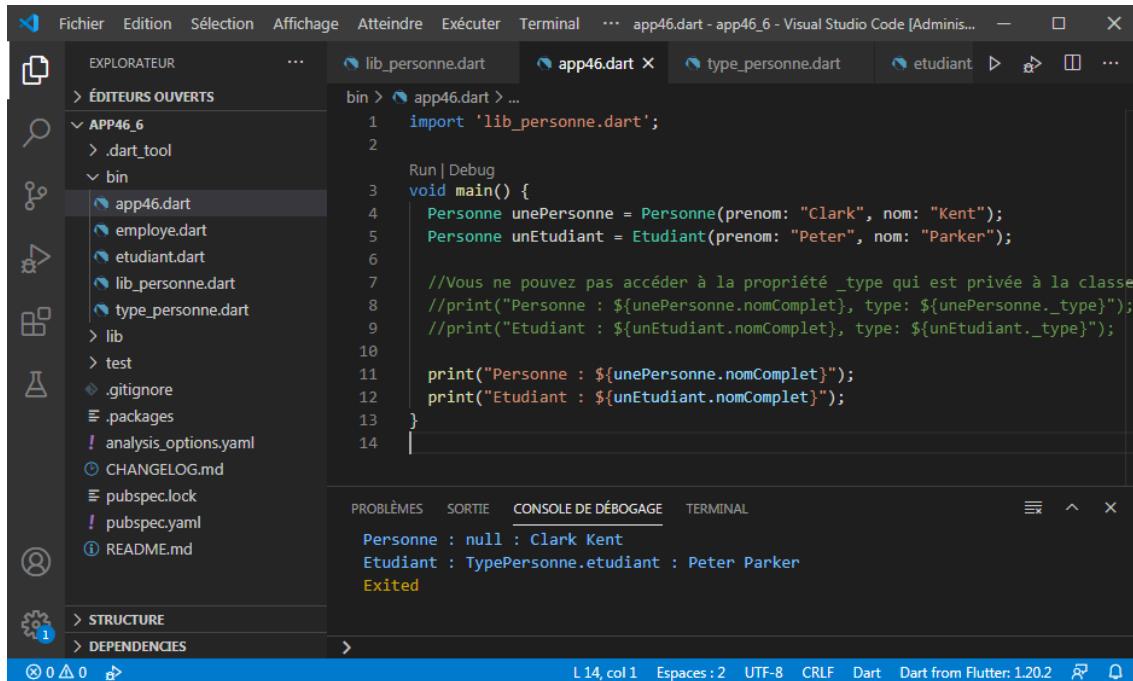


```
part of personne;

class Employe extends Personne {
    Employe([prenom, nom]) : super(prenom, nom) {
        _type = TypePersonne.employe;
}
```

En outre, comme vous pouvez le voir, la propriété `_type` est également accessible dans les fichiers associés, car elle est privée pour la bibliothèque `personne` et tous les fichiers se trouvent dans la même bibliothèque.

Jetons un coup d'œil au code suivant, qui utilise la bibliothèque `personne` :



```
Fichier Edition Sélection Affichage Atteindre Exécuter Terminal ... app46.dart - app46_6 - Visual Studio Code [Admin...]
EXPLORATEUR ...
> ÉDITEURS OUVERTS
  APP46_6
    > .dart_tool
    > bin
      app46.dart
      employe.dart
      etudiant.dart
      lib_personne.dart
      type_personne.dart
    > lib
    > test
    .gitignore
    .packages
    analysis_options.yaml
    CHANGELOG.md
    pubspec.lock
    pubspec.yaml
    README.md
  > STRUCTURE
  > DEPENDENCIES

bin > app46.dart > ...
1 import 'lib_personne.dart';
2
3 Run | Debug
4 void main() {
5   Personne unePersonne = Personne(prenom: "Clark", nom: "Kent");
6   Personne unEtudiant = Etudiant(prenom: "Peter", nom: "Parker");
7
8   //Vous ne pouvez pas accéder à la propriété _type qui est privée à la classe
9   //print("Personne : ${unePersonne.nomComplet}, type: ${unePersonne._type}");
10  //print("Etudiant : ${unEtudiant.nomComplet}, type: ${unEtudiant._type}");
11
12  print("Personne : ${unePersonne.nomComplet}");
13  print("Etudiant : ${unEtudiant.nomComplet}");
14 }

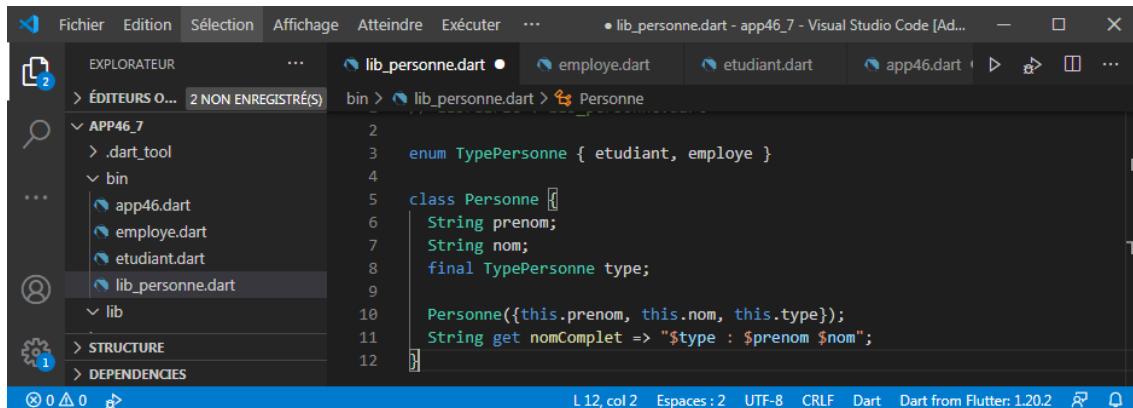
PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Personne : null : Clark Kent
Etudiant : TypePersonne.etudiant : Peter Parker
Exited

L 14, col 1 Espaces : 2 UTF-8 CRLF Dart Dart from Flutter: 1.20.2 ⚡ ⚡
```

Jetez un œil au code précédent ; le programme client de la bibliothèque `personne` n'a besoin de rien changer, car les modifications que nous avons apportées se trouvent dans la structure interne de la bibliothèque.

4.6.5.2.3 Une bibliothèque multi-fichiers - l'instruction d'exportation

L'approche précédente n'est pas la manière idéale de fractionner une bibliothèque Dart. En effet, la syntaxe de l'instruction `part` est susceptible de changer dans les versions futures. De plus, vous l'avez peut-être trouvé un peu exagéré et difficile à utiliser si vous souhaitez simplement contrôler la visibilité des membres de la bibliothèque. Nous pouvons choisir de ne pas créer les parties de la bibliothèque et de simplement diviser la bibliothèque en petites bibliothèques individuelles. Pour les exemples précédents, cela entraînerait des changements importants au cours de la mise en œuvre. Nous avons les parties précédentes sous forme de trois bibliothèques individuelles : `lib_personne`, `employe` et `etudiant`. Bien que liés, ils se comportent comme des bibliothèques individuelles et ne savent rien des autres membres publics :



```
Fichier Edition Sélection Affichage Atteindre Exécuter ... lib_personne.dart - app46_7 - Visual Studio Code [Ad...]
EXPLORATEUR ...
> ÉDITEURS O... 2 NON ENREGISTRÉ(S)
  APP46_7
    > .dart_tool
    > bin
      app46.dart
      employe.dart
      etudiant.dart
      lib_personne.dart
    > lib
    > STRUCTURE
    > DEPENDENCIES

bin > lib_personne.dart > Personne
2
3 enum TypePersonne { etudiant, employe }
4
5 class Personne {
6   String prenom;
7   String nom;
8   final TypePersonne type;
9
10  Personne({this.prenom, this.nom, this.type});
11  String get nomComplet => "$type : $prenom $nom";
12 }

L 12, col 2 Espaces : 2 UTF-8 CRLF Dart Dart from Flutter: 1.20.2 ⚡ ⚡
```

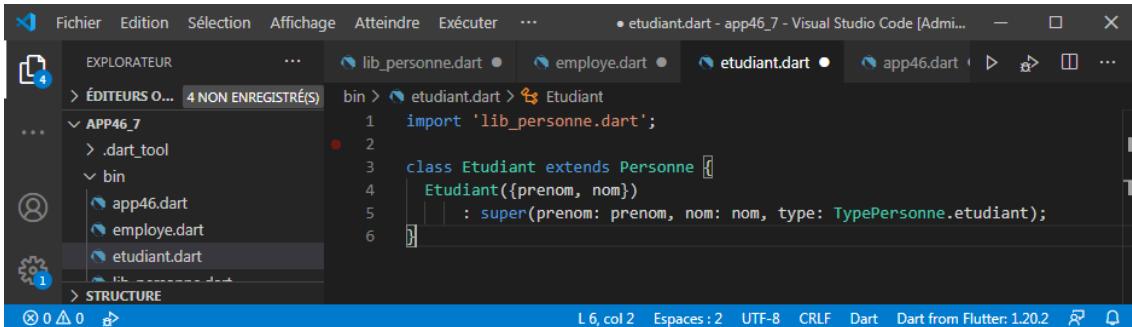
La bibliothèque `lib_personne` n'a pas besoin de l'identificateur de bibliothèque dans ce cas.

La bibliothèque de `employe` importe la bibliothèque de personnes pour accéder à sa classe `Personne`:



```
lib> employe.dart > Employe
1 import 'lib_personne.dart';
2
3 class Employe extends Personne {
4     Employe({prenom, nom})
5         : super(prenom: prenom, nom: nom, type: TypePersonne.employe);
6 }
```

De la même manière, la bibliothèque `etudiant` importe la bibliothèque `lib_personne`:

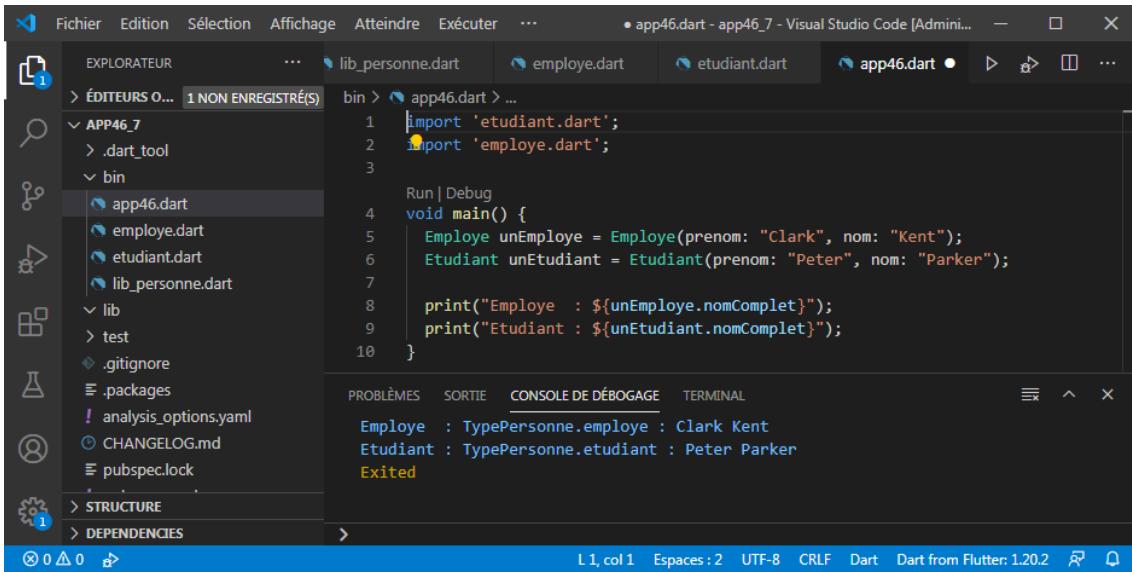


```
lib> etudiant.dart > Etudiant
1 import 'lib_personne.dart';
2
3 class Etudiant extends Personne {
4     Etudiant({prenom, nom})
5         : super(prenom: prenom, nom: nom, type: TypePersonne.etudiant);
6 }
```

Vous pouvez voir ce qui suit à partir du code précédent :

- Les bibliothèques `employe` et `etudiant` doivent importer la bibliothèque `lib_personne` pour l'étendre.
- En outre, la propriété `type` de la classe `Personne` a été rendue publique en supprimant le préfixe `'_'`. Cela signifie qu'il est accessible par les autres bibliothèques. Comme la propriété `type`, dans ce cas, n'est pas destinée à changer et qu'elle est initialisée dans le constructeur, nous l'avons également rendue définitive.

Jetons un coup d'œil au programme client de la bibliothèque, comme suit:



```
bin> app46.dart > ...
1 import 'etudiant.dart';
2 import 'employe.dart';
3
4 Run | Debug
void main() {
    Employe unEmploye = Employe(prenom: "Clark", nom: "Kent");
    Etudiant unEtudiant = Etudiant(prenom: "Peter", nom: "Parker");
}
10 }
```

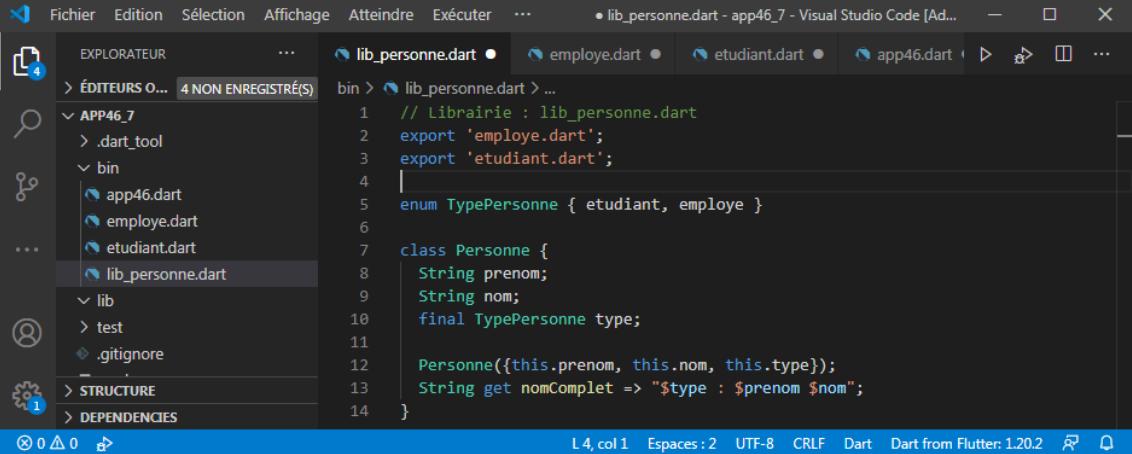
PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL

```
Employe : TypePersonne.employe : Clark Kent
Etudiant : TypePersonne.etudiant : Peter Parker
Exited
```

Le programme client de la bibliothèque aura un petit changement, car maintenant la bibliothèque est divisée en plusieurs parties, nous devrons donc importer chaque bibliothèque que nous voulons utiliser individuellement.

Ce n'est pas un gros problème quand on parle de petites bibliothèques, mais essayez de penser à une structure de bibliothèque plus complexe, où l'importation de toutes les bibliothèques interdépendantes individuellement ajouterait des difficultés à son utilisation.

C'est là que l'instruction d'exportation `export` entre en jeu. Ici, nous pouvons sélectionner le fichier de bibliothèque principal et, à partir de là, exporter toutes les petites bibliothèques qui lui sont associées. De cette façon, le client n'a besoin d'importer qu'une seule bibliothèque et toutes les plus petites bibliothèques seront disponibles à côté. Dans notre exemple, le meilleur choix pour utiliser ceci pourrait être la bibliothèque `lib_personne` :



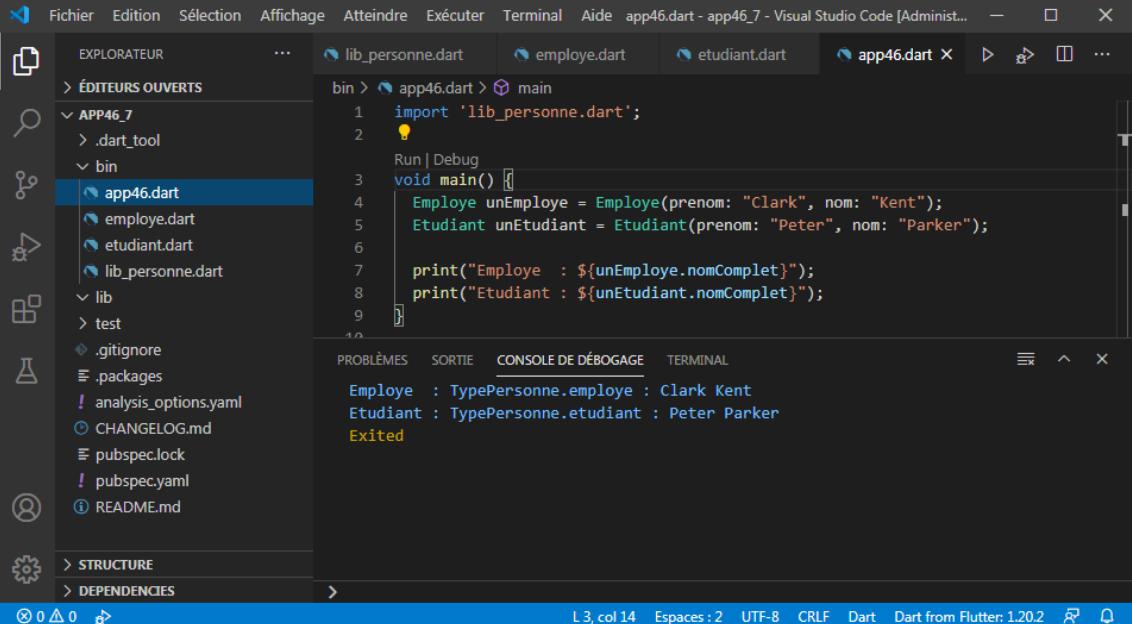
```
// Librairie : lib_personne.dart
export 'employe.dart';
export 'etudiant.dart';

enum TypePersonne { etudiant, employe }

class Personne {
  String prenom;
  String nom;
  final TypePersonne type;

  Personne({this.prenom, this.nom, this.type});
  String get nomComplet => "$type : $prenom $nom";
}
```

De cette façon, le programme client de la bibliothèque serait comme suit:



```
import 'lib_personne.dart';

void main() {
  Employe unEmploye = Employe(prenom: "Clark", nom: "Kent");
  Etudiant unEtudiant = Etudiant(prenom: "Peter", nom: "Parker");

  print("Employe : ${unEmploye.nomComplet}");
  print("Etudiant : ${unEtudiant.nomComplet}");
}

Employe : TypePersonne.employe : Clark Kent
Etudiant : TypePersonne.etudiant : Peter Parker
Exited
```

Notez que seule l'instruction d'importation change. Nous pouvons utiliser les classes des petites bibliothèques normalement car elles sont exportées depuis `lib_personne`. Après avoir compris le concept de bibliothèque Dart, nous pouvons maintenant examiner comment combiner ces morceaux de code en quelque chose de partageable et réutilisable : le package Dart.

4.6.6 Les packages Dart

4.6.6.1 Introduction

Un package Dart est le point de départ de tout projet Dart. Dans les exemples précédents, nous ne nous en préoccupons pas car nous utilisions des exemples de syntaxe à fichier unique ; cependant, dans le monde réel, nous travaillerons toujours avec des packages ou encore des regroupements de bibliothèques.

Le principal avantage de l'utilisation et de la création de packages est que le code peut être réutilisé et partagé. Dans l'écosystème Dart, cela est fait par l'outil `pub`, qui nous permet d'extraire et d'envoyer des dépendances au site Web contenant le référentiel « pub.dartlang.org ».

L'utilisation d'un package de bibliothèque dans un projet en fait une dépendance immédiate, et la bibliothèque utilisée peut avoir ses propres dépendances, appelées dépendances transitives. En général, il existe deux types de packages Dart : les packages d'application et les packages de bibliothèque.

4.6.6.2 Les packages d'application et les packages de bibliothèque

Tous les packages ne sont pas censés être partageables ; une application elle-même est également un package. Ces packages peuvent normalement avoir des dépendances sur les packages de bibliothèque, mais ils ne sont pas destinés à être utilisés comme dépendance dans d'autres projets.

D'autre part, les packages de bibliothèque sont ceux qui contiennent du code utile qui peut être utile dans de nombreux projets. Ces packages peuvent être utilisés comme dépendance et ont également des dépendances sur d'autres.

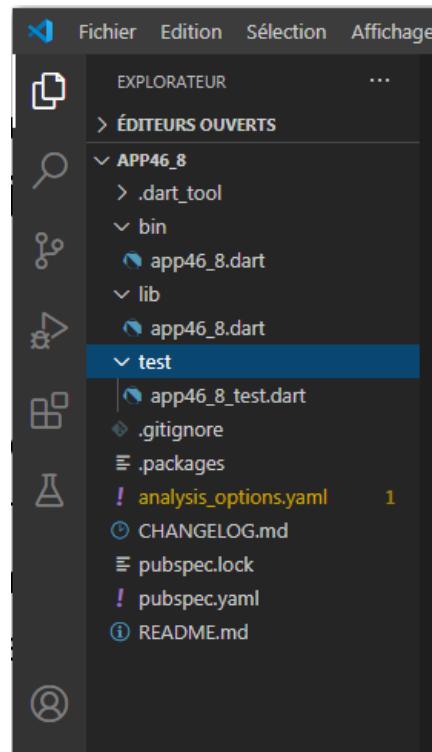
En termes simples, la structure recommandée d'un package Dart ne diffère pas trop entre une application et un package de bibliothèque - leur objectif et leur utilisation sont différents les uns des autres.

4.6.6.3 La structure de package

La première chose importante à souligner à propos d'une structure de package application Dart est que sa validité est contrôlée par la présence d'un fichier **`pubspec.yaml`** ; autrement dit, s'il y a un fichier **`pubspec.yaml`** dans votre structure, alors il y a un paquet et c'est là que vous le décrivez correctement - sans lui, il n'y a pas du tout de paquet.

Voici à quoi ressemble un package typique =>

NB : Cet exemple de package a été généré à l'aide de l'outil « Stagehand » que vous avez incorporé dans Visual Studio Code.



Pour les packages d'application, aucune mise en page de projet n'est requise (car elle n'est pas destinée à être publiée dans le référentiel pub). Cependant, au fur et à mesure de son évolution, il existe déjà plusieurs méthodes et conventions à suivre. Jetons un coup d'œil à la structure commune d'un package Dart général. La plus grande partie de cette structure est conventionnelle et dépend de la complexité de votre projet et si vous souhaitez partager son code d'une manière ou d'une autre.

Jetons un coup d'œil au rôle de chaque fichier et répertoire dans une structure de package Dart typique:

- ⊕ ***pubspec.yaml*** : Comme déjà souligné, il s'agit du fichier fondamental du package et il décrit le package dans le référentiel pub. Nous examinerons la structure complète de ce fichier ultérieurement.
- ⊕ Les répertoires ***lib/*** : c'est l'endroit où réside le code source de la bibliothèque de paquets. Comme vous le savez déjà, un simple fichier .dart est une petite bibliothèque, donc tout ce que vous mettez dans le répertoire lib est accessible au public pour les autres packages. C'est ce qu'on appelle l'API publique du package.
- ⊕ ***test/*** : les tests unitaires sont classiquement placés dans le répertoire ***test***. En outre, le code source à l'intérieur du dossier de test est généralement postfixé avec l'identificateur _test.
- ⊕ ***README.md*** et ***CHANGELOG.md*** : ce sont des fichiers de présentation généralement présents dans des packages destinés à être publiés dans un référentiel public, tel que le pub Dart. Ces fichiers sont également très courants dans les projets open source.
- ⊕ ***analysis_options.yaml*** : il s'agit d'un fichier utile pour personnaliser les contrôles, l'analyse de style et d'autres contrôles de précompilation.

Dans les packages Web, de nouveaux fichiers et répertoires sont inclus tels que :

- ⊕ Le répertoire ***lib/*** est la destination typique des fichiers de ressources Web statiques, tels que des images ou des fichiers .css.
- ⊕ Le répertoire ***web/*** est un répertoire utilisé dans les projets d'application Web. Contrairement au dossier ***lib/***, qui est censé être du code de bibliothèque, ce code est censé avoir le code source et les points d'entrée de l'application Web (c'est-à-dire la fonction main()).

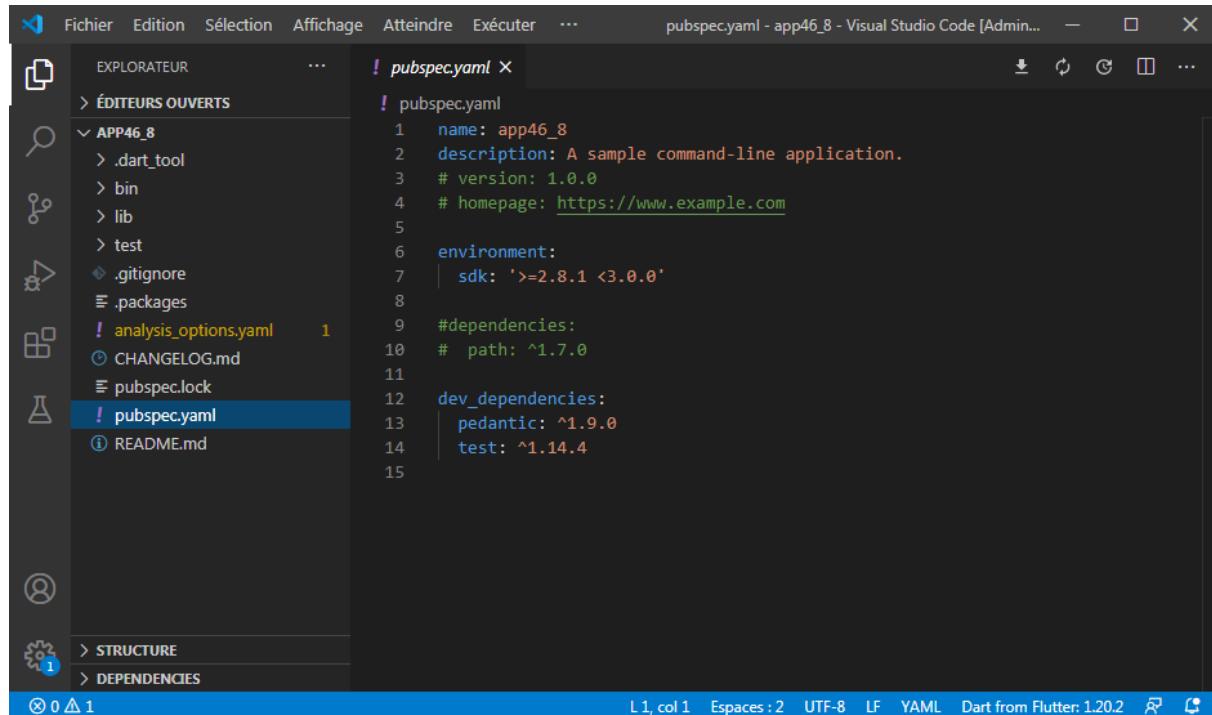
Dans les packages de ligne de commande, le répertoire ***bin/*** est inclus :

- ⊕ Le répertoire ***bin/*** est censé avoir un script qui peut s'exécuter directement à partir de la ligne de commande.

4.6.6.4 *Le fichier pubspec.yaml*

Le fichier ***pubspec.yaml*** est au cœur d'un package Dart, et pour comprendre comment décrire correctement le package, nous devons comprendre comment ce fichier est structuré. Ce fichier est basé sur la syntaxe ***yaml***, un format commun utilisé pour les fichiers de configuration, avec une structure facile à lire et à suivre.

Le fichier ***pubspec.yaml*** est le suivant:



```
! pubspec.yaml x
! pubspec.yaml
1 name: app46_8
2 description: A sample command-line application.
3 # version: 1.0.0
4 # homepage: https://www.example.com
5
6 environment:
7 | sdk: '>=2.8.1 <3.0.0'
8
9 dependencies:
10 | # path: ^1.7.0
11
12 dev_dependencies:
13 | pedantic: '^1.9.0'
14 | test: '^1.14.4'
15
```

Le fichier spécifie les informations de métadonnées du package, ce qui est utile si vous souhaitez publier le package. Il définit également les dépendances tierces du package et la version du SDK Dart. Examinons les champs de ***pubspec.yaml*** plus en détail:

- + ***name*** : Il s'agit de l'identifiant du package. Il est obligatoire et ne doit contenir que des lettres minuscules et des chiffres, plus le caractère '_'; en outre, il doit s'agir d'un identifiant Dart valide (c'est-à-dire qu'il ne peut pas commencer par des chiffres et ne peut pas être un mot réservé). C'est une propriété très importante si vous souhaitez publier le package dans le référentiel pub, et il est bon de vérifier les noms de packages existants pour éviter la duplication.
- + ***description*** : bien qu'il s'agisse d'un champ facultatif, il est obligatoire si vous avez l'intention de publier le package, en décrivant en termes simples l'objectif du package.
- + ***version*** : Ceci est également facultatif pour les packages personnels, mais il est requis pour la publication dans le référentiel pub. Il est important de maintenir la cohérence dans la gestion des versions d'un package qui sera utilisable par la communauté.
- + ***homepage*** : pour les packages pub, cela sera lié à la page du package sur le site Web de publication. Il est très important d'en fournir une lorsque vous prévoyez de le publier.
- + ***authors* ou *authors*** : Bien que cela ne soit pas obligatoire, il est important de fournir des informations de contact sur le ou les créateurs de la bibliothèque. De plus, une bibliothèque peut avoir plus d'un auteur; dans ce cas, la syntaxe de la liste YAML peut être utilisée en définissant le champ auteurs à la place (notez les informations de contact facultatives):

```
authors :
- Alexis GUILLOTEAU <alexis.guilloteau@imt-lille-douai.fr>
- Didier JUGE-HUBERT <didier.juge-hubert@imt-lille-douai.fr>
```

- dependencies et dev_dependencies** : elles se réfèrent à l'objectif réel du fichier **pubspec**. Une liste des packages tiers est requise pour l'utilisation de la bibliothèque et respectivement le développement de la bibliothèque.,
 - environnement** : Outre les dépendances tierces, il existe une autre, disons, la dépendance principale d'un package, qui est le SDK Dart lui-même. Dans ce champ, vous devez spécifier la cible et les versions prises en charge du SDK Dart.
- NB** : Le champ d'environnement est la dépendance du SDK; il est recommandé de spécifier la version cible du SDK Dart en utilisant la syntaxe de plage, car la notation sémantique de plage n'est pas compatible avec les anciennes versions (c'est-à-dire <1.8.3).

La structure **pubspec** typique contient les champs spécifiés précédemment. Pour une explication complète du fichier **pubspec** et des autres champs spécifiques à un objectif, consultez le site Web de Dart: <https://www.dartlang.org/tools/pub/pubspec>.

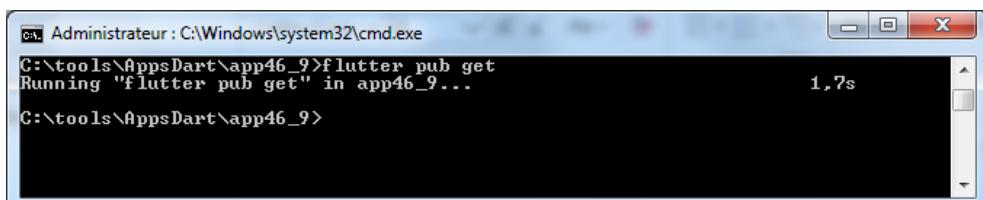
4.6.6.5 Les dépendances de package – flutter pub

Maintenant que vous comprenez le rôle le plus important du fichier **pubspec** dans le package lors du développement d'applications Dart, vous pouvez ajouter des dépendances de package tiers à votre projet. Il existe une commande importante : **pub** avec laquelle vous pouvez travailler lors de l'ajout ou de la mise à jour de dépendances de package dans votre projet. Nous devons également montrer comment spécifier correctement la version de dépendance que nous devons utiliser.

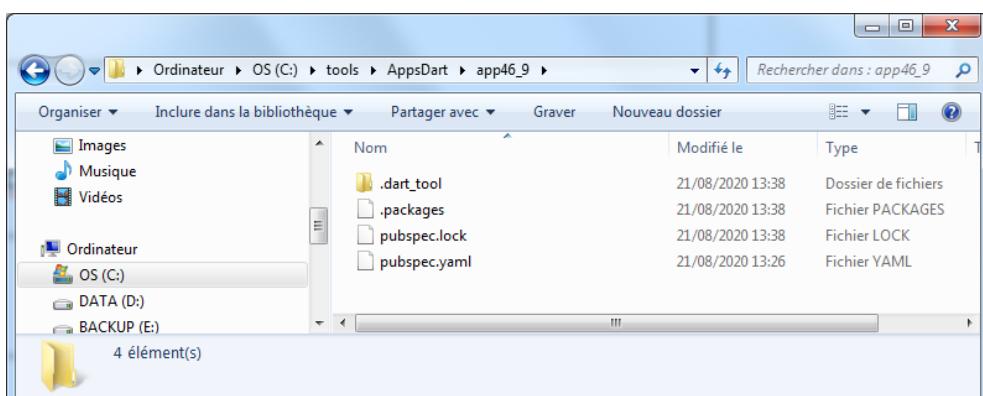
Après avoir démarré un nouveau projet Dart, manuellement ou à l'aide d'un outil générateur tel que « Stagehand », la première chose à faire est d'éditer le fichier **pubspec.yaml** minimal qui contient la ligne suivante :

```
name: adding_dependencies
```

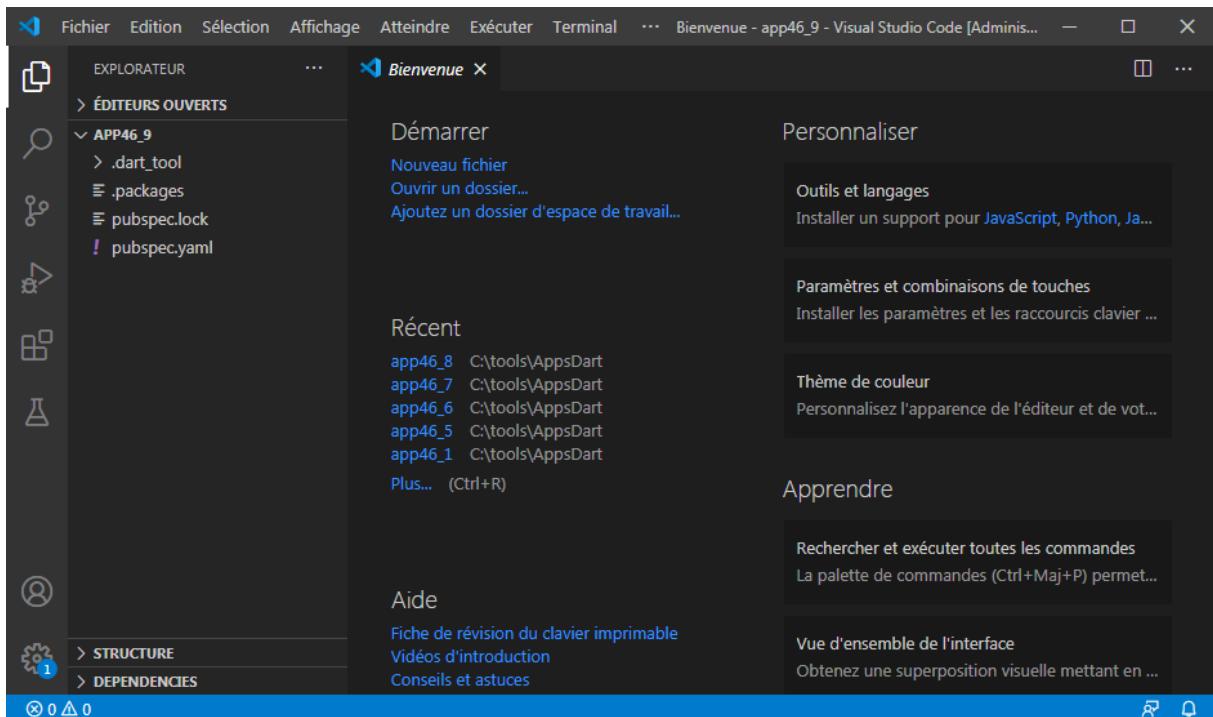
Il s'agit d'une description minimale du package et il n'a aucune dépendance spécifiée, pas même la version cible du SDK Dart. Cependant, exécutons la commande « **flutter pub get** » dans le dossier du package, car elle fonctionnera de la même manière:



Nous obtenons dans le répertoire :



Vous avez des dépendances ! Nous obtiendrons une structure de fichiers comme dans la capture d'écran suivante :



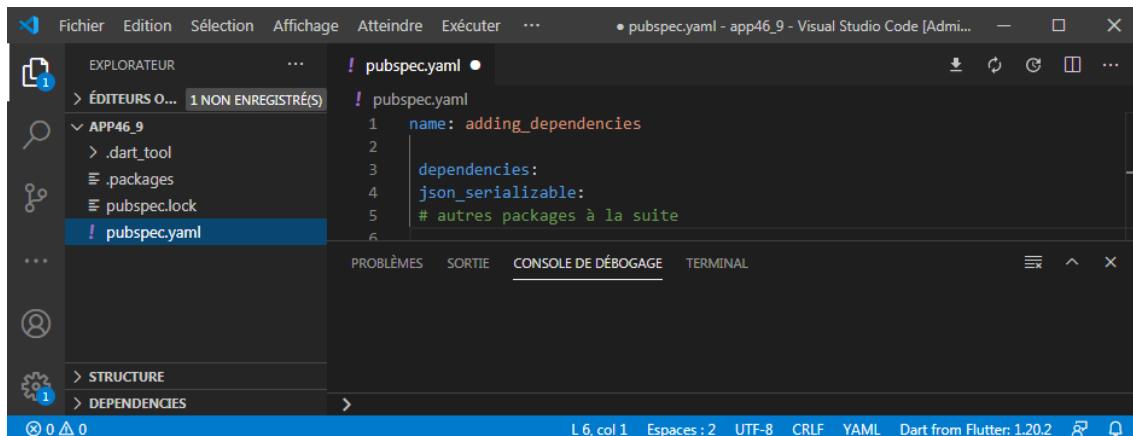
Notez les nouveaux fichiers générés par la commande dans le répertoire. Ces fichiers sont importants pour que l'outil pub fonctionne avec les packages de dépendances :

- ⊕ **.packages** : Ceci mappe les dépendances dans le cache pub du système. Au lieu de faire des copies dans tous vos packages, l'outil pub stocke simplement le mappage entre le package et son emplacement respectif dans le système. Une fois le package mappé ici, vous pourrez l'importer dans votre code Dart. Ce fichier ne doit pas être inclus dans le système de gestion du code source; c'est parce qu'il est généré et géré par l'outil **pub**.
- ⊕ **pubspec.lock**: C'est le fichier auxiliaire de l'outil **pub** qui contient tous les graphiques de dépendances du paquet, c'est-à-dire qu'il répertorie toutes les dépendances immédiates et les dépendances transitives. Il contient également les versions exactes et d'autres informations de métadonnées sur toutes les dépendances. Il est recommandé d'inclure ce fichier dans le système de gestion des sources uniquement s'il s'agit d'un package d'application. Cela aide une équipe de développement, par exemple, à travailler avec exactement la même configuration de dépendances. Si vous utilisez un package de bibliothèque, il n'est généralement pas inclus, car il devrait fonctionner avec une large gamme de dépendances, c'est-à-dire qu'il ne doit pas être verrouillé sur des versions spécifiques.

4.6.6.6 Spécification des dépendances

Maintenant que vous savez comment l'outil **pub** résout les packages à l'intérieur du projet, voyons comment y ajouter des dépendances. Les dépendances sont spécifiées dans le champ *dependencies* du fichier **pubspec.yaml**. Il s'agit d'un champ de liste YAML, vous pouvez donc en spécifier autant que nécessaire dans le champ. Supposons que nous ayons besoin du package *json_serializable* dans notre projet.

Nous pouvons le spécifier en l'ajoutant simplement à la liste, comme suit :



```
name: adding_dependencies
dependencies:
  json_serializable:
    # autres packages à la suite
```

La syntaxe pour spécifier une dépendance est la suivante:

```
<package>: <constraints>
```

Ici, vous ajoutez son nom (<package>) suivi des champs <constraints>: version et source. Dans ce cas, nous n'avons spécifié aucune contrainte, donc cela suppose une version disponible pour la contrainte de version et la source par défaut (pub.dartlang.org).

4.6.6.7 Spécification des contraintes des dépendances

4.6.6.7.1 La version

La contrainte de version peut être un numéro de version simple, une plage ou une contrainte minimale ou maximale. Explorons à quoi cela ressemble dans chaque situation :

- + any/empty : Comme l'exemple précédent, nous pouvons l'utiliser sans contrainte de version,

```
json_serializable:
# ou
json_serializable:any.
```

- + Version précise : nous pouvons ajouter le numéro de version spécifique avec lequel nous voulons travailler

```
json_serializable:2.0.1.
```

- + Limite minimale : Ici, nous pouvons ajouter une version minimale acceptable du package que nous voulons de deux manières:

```
json_serializable:> 1.0.0'
# où nous acceptons toute version ultérieure à la version spécifiée exclue
# ou
json_serializable:> = 1.0.0'
# où nous acceptons toute version supérieure ou égale à la version spécifiée.
```

- Limité maximale : Comme l'exemple minimum précédent mais dans la limite supérieure, nous pouvons ajouter une version maximale acceptable du package que nous voulons de deux manières :

```
json_serializable:'<2.0.1'
# où nous acceptons toute version inférieure à celle spécifiée,
# ou
json_serializable:'<= 2.0.1'
# où nous acceptons toute version inférieure ou égale à celle spécifiée.
```

- Plage : en combinant des limites minimales et maximales, nous pouvons spécifier un intervalle acceptable de versions (l'une des quatre) :

```
json_serializable:'> 1.0.0 <= 2.0.1'
json_serializable:'> 1.0.0 <2.0.1'
json_serializable:'> = 1.0.0 <2.0.1'
json_serializable:'> = 1.0.0 <= 2.0.1'
```

- Plage sémantique : Ceci est similaire à plage mais, en utilisant le caractère '^', nous pouvons spécifier la plage entre une version minimale acceptable et la prochaine modification de rupture.

```
json_serializable: ^1.0.0
# est identique à json_serializable: '>=1.0.0 <2.0.0'
json_serializable: ^ 0.1.0
# est identique à json_serializable: '>=0.1.0 <0.2.0'
```

4.6.6.7.2 La contrainte source

L'outil **pub** ne recherche pas uniquement dans le référentiel pub les packages. Si vous avez déjà utilisé un autre système de gestion de packages, vous savez qu'il est parfois utile d'héberger vos packages dans d'autres endroits que le référentiel public, tels que les packages privés de l'entreprise ou ceux de votre utilisation personnelle.

Pour la partie source de la spécification du package, nous avons quatre alternatives pour changer l'endroit où l'outil **pub** doit rechercher le package :

Source hébergée : il s'agit du dépôt pub par défaut ou d'un autre serveur http alternatif qui implémente l'api pub.

```
dependencies:
  json_serializable:
    hosted:
      name: json_serializable
      url: http://pub-packages-private-server.com # serveur à modifier
```

Comme vous pouvez le voir, nous n'avons besoin de spécifier le champ hébergé que si nous n'utilisons pas le dépôt pub, c'est-à-dire la source par défaut.

Source locale : Ici, vous pouvez ajouter une dépendance à un package dans votre propre système :

```
dependencies:
  json_serializable:
    path: C:\tools\packages\json_serializable
```

Bien que vous ne soyez pas autorisé à partager un package avec ce type de dépendance, cela peut être utile dans les étapes de développement.

Source Git : Ici, vous pouvez spécifier un package à partir d'un dépôt git :

```
dependencies:  
  json_serializable:  
    git:  
      url: git: //github.com/dart-lang/json_serializable.git  
      path: path/to/json_serializable # si la racine du package est  
          # pas la racine du  
          # référentiel  
      ref: master # dépendant d'un commit, d'une balise, d'une branche spécifique
```

Cela peut être utile dans les étapes de développement ou si un code source de paquet publié n'est pas encore présent dans le référentiel pub.

Source SDK : Un SDK peut avoir ses propres packages qui peuvent être utilisés comme dépendances :

```
dependencies:  
  flutter_localizations: # une dépendance disponible dans le sdk flutter  
  sdk: flutter
```

Jusqu'à présent, cette façon de spécifier les contraintes de source n'est utilisée que pour les dépendances du SDK Flutter.

Les dépendances de package sont un sujet fondamental dans le développement de Dart. Avec ces concepts à l'esprit, vous pouvez ajouter des dépendances tierces utiles à vos projets et augmenter votre productivité.

4.7 La programmation asynchrone avec *Future* et *Isolate*

4.7.1 Introduction

Dart est un langage de programmation mono thread, c'est-à-dire que tout le code de l'application s'exécute dans le même thread. En termes simples, cela signifie que n'importe quel code peut bloquer l'exécution du thread en exécutant des opérations de longue durée telles que des demandes d'E/S ou http://.

Bien que Dart soit mono thread, il peut effectuer des opérations asynchrones via l'utilisation de **Future**. En outre, pour représenter le résultat de ces opérations asynchrones, Dart utilise l'objet **Future** combiné avec les mots clés **async** et **await**. Comprendre ces concepts importants pour développer une application réactive.

4.7.2 Future

L'objet **Future<T>** dans Dart représente une valeur qui sera fournie ultérieurement. Il peut être utilisé pour marquer une méthode, par exemple, avec un résultat futur ; c'est-à-dire qu'une méthode retournant un objet **Future<T>** n'aura pas la valeur de résultat appropriée immédiatement, mais à la place après un certain calcul à un moment ultérieur.

Prenons un exemple et considérons le code suivant, où nous avons la fonction principale qui appelle une opération de longue durée.

```

Fichier Edition Sélection Affichage Atteindre Exécuter ...
app46.dart - app46_10 - Visual Studio Code [Admin...]
EXPLORATEUR
> ÉDITEURS OUVERTS
APP46_10
> .dart_tool
> bin
> app46.dart
> lib
> test
> .gitignore
> .packages
! analysis_options.yaml
CHANGELOG.md
pubspec.lock
! pubspec.yaml
README.md

app46.dart
bin > app46.dart > ...
1 import 'dart:io';
2
3 void operationLongue() {
4     for (int i = 0; i < 5; i++) {
5         sleep(Duration(seconds: 1));
6         print("index: $i");
7     }
8 }
9
Run | Debug
10 main() {
11     print("Début de l'opération longue");
12     operationLongue();
13     print("Continuation du programme principal");
14
15     for (int i = 10; i < 15; i++) {
16         sleep(Duration(seconds: 1));
17         print("index principal: $i");
18     }
19
20     print("Fin du programme principal");
21 }

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Début de l'opération longue
index: 0
index: 1
index: 2
index: 3
index: 4
Continuation du programme principal
index principal: 10
index principal: 11
index principal: 12
index principal: 13
index principal: 14
Fin du programme principal
Exited

```

Si vous exécutez le code précédent, vous remarquerez qu'il arrête l'exécution de la fonction principale pendant que la fonction `operationLongue()` est en cours d'exécution. Il s'agit d'une exécution synchrone de tout le code et elle ne conviendra probablement pas à tous les cas d'utilisation. Maintenant, disons que la fonction `operationLongue()` est une fonction asynchrone et la fonction principale : `main()` peut continuer à s'exécuter sans attendre la fin pour continuer:

```

Fichier Edition Sélection Affichage Atteindre Exécuter ...
app46.dart - app46_10 - Visual Studio Code [Admin...]
EXPLORATEUR
> ÉDITEURS OUVERTS
APP46_10
> .dart_tool
> bin
> app46.dart
> lib
> test
> .gitignore
> .packages
! analysis_options.yaml
CHANGELOG.md
pubspec.lock
! pubspec.yaml
README.md

app46.dart
bin > app46.dart > operationLongue
1 import 'dart:io';
2
3 Future operationLongue() async {
4     for (int i = 0; i < 5; i++) {
5         await Future.delayed(Duration(seconds: 1));
6         print("index: $i");
7     }
8 }
9
Run | Debug
main() {
    print("Début de l'opération longue");
    operationLongue();
    print("Continuation du programme principal");
}
10
11 for (int i = 10; i < 15; i++) {
12     sleep(Duration(seconds: 1));
13     print("index principal: $i");
14 }
15
16 print("Fin du programme principal");
17 }

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Début de l'opération longue
Continuation du programme principal
index principal: 10
index principal: 11
index principal: 12
index principal: 13
index principal: 14
Fin du programme principal
index: 0
index: 1
index: 2
index: 3
index: 4
Exited

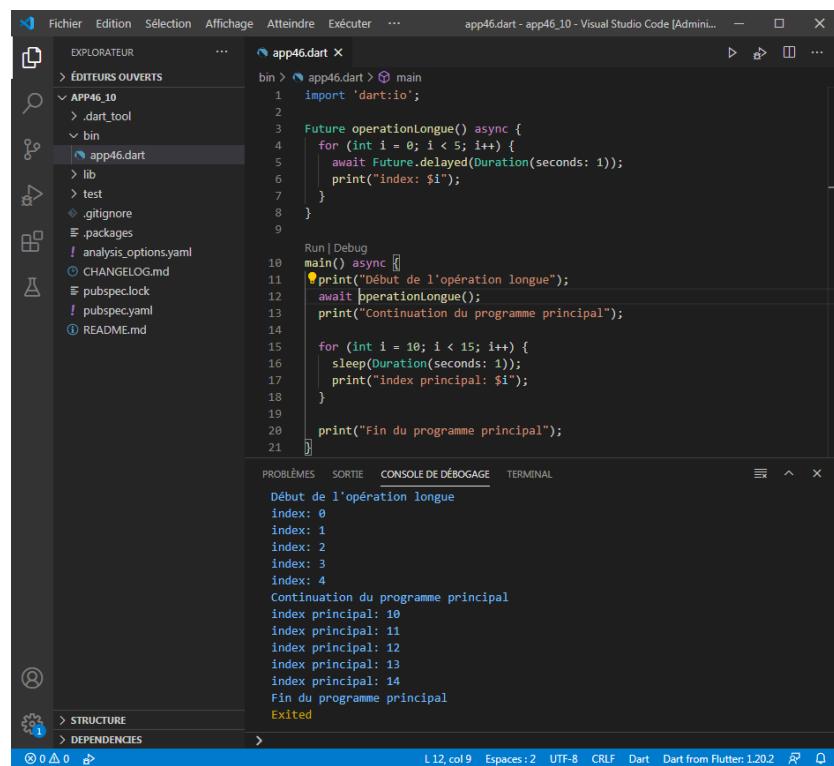
```

Nous avons apporté quelques modifications pour démontrer comment `Future` fonctionne correctement : La fonction `operationLongue()` a maintenant le modificateur `async` pour

indiquer que cela retournera une fonction **Future** et que la fonction **Future** sera terminée à la fin de l'exécution de la fonction. Notez que le type de retour est également **Future**. Nous avons remplacé l'appel `sleep()` par l'appel `Future.delayed`. Cela démontre l'utilisation du mot-clé `await`. Le mot clé `await` fonctionne avec les fonctions asynchrones. Lors de l'appel d'une fonction **Future**, nous pouvons avoir besoin du résultat de la fonction **Future** pour continuer l'exécution. Dans ce cas, nous souhaitons procéder à l'impression uniquement après le délai spécifié.

Vous pouvez voir que le résultat de l'exécution de ce programme peut-être quelque chose d'étrange. Ce n'est pas un code concurrent où un code s'exécute après un autre comme avant ; ici, ce qui change, c'est l'ordre.

Dans l'exemple précédent, la modification se produit lorsque les appels de la fonction `operationLongue()` attendent dans une autre fonction asynchrone. Ici, la fonction est suspendue et ne sera reprise qu'après un délai de 1 seconde. Après le délai, cependant, la fonction principale est déjà en cours d'exécution car elle n'attend plus la fin de l'opération longue, de sorte que le code `operationLongue()` ne sera exécuté qu'une fois la fonction principale terminée. Une chose que nous pouvons faire est de transformer la fonction `main()` en fonction asynchrone et d'attendre l'exécution de `operationLongue()`. De cette façon, la fonction `main()` sera suspendue dès que nous appelons `wait` `operationLongue()` et ne sera reprise qu'après son exécution. Cela se comporte comme un code synchrone normal, comme suit :



```
Fichier Edition Sélection Affichage Atteindre Exécuter ...
app46.dart - app46_10 - Visual Studio Code [Admin...]
ÉDITEURS OUVERTS
APP46_10
> .dart_tool
bin
> app46.dart
> lib
> test
> .gitignore
.packages
analysis_options.yaml
CHANGELOG.md
pubspec.lock
pubspec.yaml
README.md

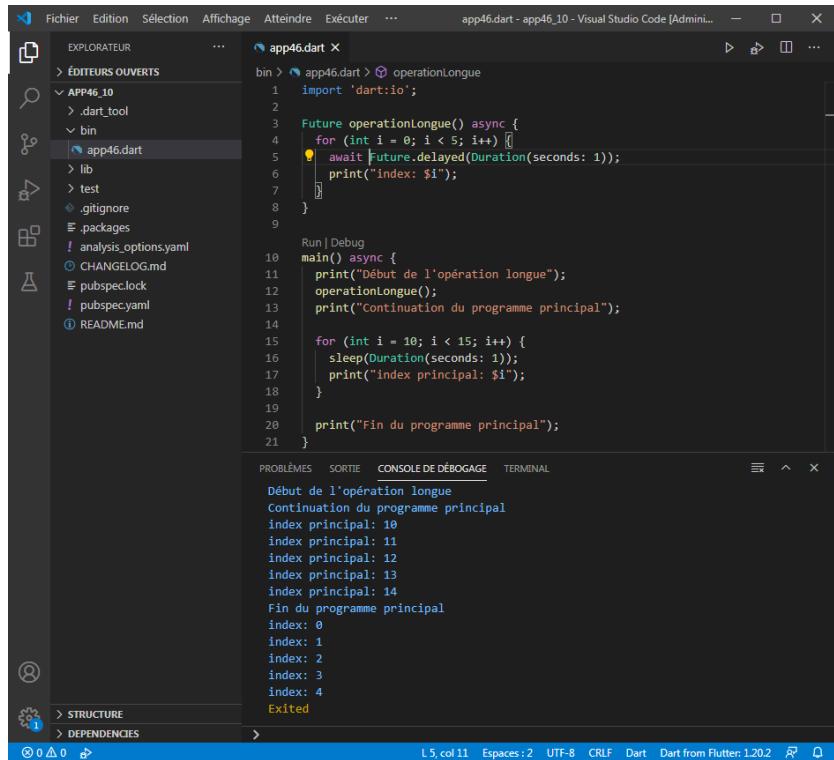
app46.dart
bin > app46.dart > main
1 import 'dart:io';
2
3 Future operationLongue() async {
4     for (int i = 0; i < 5; i++) {
5         await Future.delayed(Duration(seconds: 1));
6         print("Index: $i");
7     }
8 }
9
Run | Debug
10 main() async {
11     print("Début de l'opération longue");
12     await operationLongue();
13     print("Continuation du programme principal");
14
15     for (int i = 10; i < 15; i++) {
16         sleep(Duration(seconds: 1));
17         print("Index principal: $i");
18     }
19
20     print("Fin du programme principal");
21 }

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Début de l'opération longue
index: 0
index: 1
index: 2
index: 2
index: 3
index: 4
Continuation du programme principal
index principal: 10
index principal: 11
index principal: 12
index principal: 13
index principal: 14
Fin du programme principal
Exited

STRUCTURE >
> DEPENDENCIES >
0 △ 0 ⌂
L 12, col 9 Espaces : 2 UTF-8 CRLF Dart Dart from Flutter: 1202 ⌂ ⌂
```

Comme vous l'avez peut-être remarqué, les fonctions précédentes ne s'exécutent jamais vraiment de manière asynchrone; c'est parce que nous attendons l'exécution de la méthode `operationLongue()` avant d'exécuter le reste de son code.

Pour les faire fonctionner de manière asynchrone, nous devons omettre le mot clé `await`, comme suit :

A screenshot of Visual Studio Code showing a Dart file named "app46.dart". The code contains two asynchronous operations: one using `Future.delayed` and another using a `for` loop with `sleep`. The "CONSOLE DE DÉBOGAGE" tab shows the execution output, which includes the start of the long operation, the continuation of the main program, and the completion of both operations in sequence.

```
import 'dart:io';
Future operationLongue() async {
  for (int i = 0; i < 5; i++) [
    await Future.delayed(Duration(seconds: 1));
    print("index: $i");
  ]
}

main() async {
  print("Début de l'opération longue");
  operationLongue();
  print("Continuation du programme principal");

  for (int i = 10; i < 15; i++) {
    sleep(Duration(seconds: 1));
    print("index principal: $i");
  }
  print("Fin du programme principal");
}

index: 0
index: 1
index: 2
index: 3
index: 4
Fin de l'opération longue
Continuation du programme principal
index principal: 10
index principal: 11
index principal: 12
index principal: 13
index principal: 14
Fin du programme principal
index: 0
index: 1
index: 2
index: 3
index: 4
Exited
```

Dart exécute les deux méthodes asynchrones dans le même thread. Les deux fonctions s'exécutent de manière asynchrone dans ce cas, mais cela ne signifie pas qu'elles sont exécutées en parallèle. Dart exécute une opération à la fois ; tant qu'une opération est en cours d'exécution, elle ne peut être interrompue par aucun autre code Dart. Cette exécution est contrôlée par la boucle *Dart Event*, qui agit comme un gestionnaire pour *Dart Futures* et le code asynchrone.

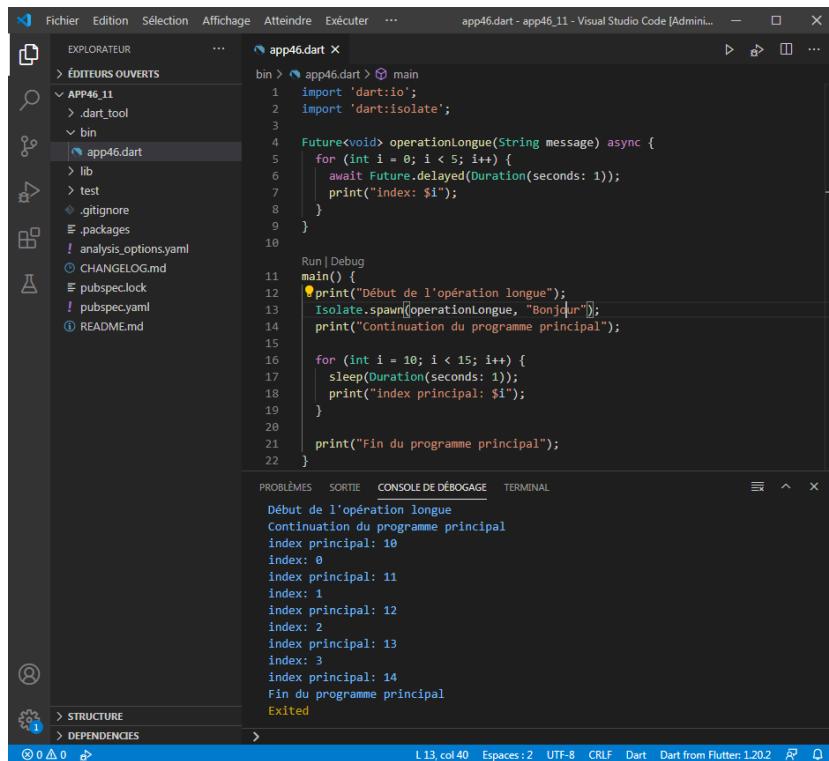
Vous pouvez vous référer à la documentation officielle de Dart sur la boucle d'événements pour comprendre comment cela fonctionne: <https://dart.dev/articles/archive/event-loop>.

4.7.3 Isolates

Alors, vous vous demandez peut-être comment exécuter du code vraiment parallèle et améliorer les performances et la réactivité ? *Isolates* sont là pour cela. Chaque application Dart est composée d'au moins une instance *Isolate*, l'instance *Isolate* principale, où tout le code de l'application s'exécute. Donc, pour créer un code d'exécution parallèle, nous devons créer une nouvelle instance *Isolate* qui peut s'exécuter en parallèle avec *Isolate* principal:

Les isolats peuvent être considérés comme une sorte de thread, mais ils ne partagent rien entre eux, comme le nom suggère. Cela signifie qu'ils ne partagent pas la mémoire, nous n'avons donc pas besoin d'utiliser des verrous et d'autres techniques de synchronisation de thread ici. Pour communiquer entre les isolats, c'est-à-dire pour envoyer et recevoir des données entre eux, nous devons échanger des messages. Dart fournit un moyen d'accomplir cela.

Modifions l'implémentation précédente pour utiliser une instance `Isolate` à la place:



The screenshot shows the Visual Studio Code interface with the following details:

- Explorateur (Left):** Shows the project structure with files like `APP46_11`, `.dart_tool`, `bin`, and `app46.dart`.
- Code Editor (Center):** Displays the `app46.dart` file content:

```
bin > app46.dart > main
1 import 'dart:io';
2 import 'dart:isolate';
3
4 Future<void> operationLongue(String message) async {
5   for (int i = 0; i < 5; i++) {
6     await Future.delayed(Duration(seconds: 1));
7     print("index: $i");
8   }
9
10
11 Run | Debug
12 main() {
13   print("Début de l'opération longue");
14   Isolate.spawn(operationLongue, "Bonjour");
15   print("Continuation du programme principal");
16
17   for (int i = 10; i < 15; i++) {
18     sleep(Duration(seconds: 1));
19     print("index principal: $i");
20   }
21
22   print("Fin du programme principal");
23 }
```
- Console (Bottom):** Shows the execution output:

```
Début de l'opération longue
Continuation du programme principal
index principal: 10
index: 0
index principal: 11
index: 1
index principal: 12
index: 2
index principal: 13
index: 3
index principal: 14
Fin du programme principal
Exited
```
- Status Bar (Bottom):** Shows the current file is `app46.dart` at line 13, column 40, with encoding UTF-8, and Dart from Flutter 1.20.2.

Comme vous pouvez le voir, le code contient de petites modifications :

La fonction `operationLongue()` devient une instance `Isolate`, c'est-à-dire qu'elle reste comme une fonction simple. Pour envoyer le processus `Isolate` à l'exécution, nous utilisons la méthode `spawn()` de la classe `Isolate`. Il prend deux arguments : la fonction à générer et un paramètre à passer à la fonction. En exécutant le code précédent, vous noterez une sortie d'exécution différente.

Désormais, le code de ces deux fonctions s'exécute indépendamment après la génération `Isolate`.

4.8 Présentation des tests unitaires avec Dart

4.8.1 Introduction

Dans n'importe quel langage, nous pouvons écrire du code qui remplit un but. Cependant, pour écrire un code performant et sans bogue, nous devons utiliser toutes les ressources disponibles. Les tests unitaires sont l'une des choses qui peuvent nous aider à écrire du code modulaire, efficace et sans bogue.

Le test unitaire n'est pas le seul moyen de tester le code, bien sûr, mais c'est une partie cruciale du test de petits logiciels de manière à l'isoler des autres parties, ce qui nous aide à nous concentrer sur des choses spécifiques.

Couvrir tout le code de l'application avec des tests unitaires ne garantit pas qu'il soit 100% exempt de bogues. Cependant, cela nous aide à atteindre progressivement un code mature, et c'est l'une des étapes pour assurer un bon cycle de développement, avec des versions stables de temps en temps.

Dart fournit également des outils utiles pour travailler avec des tests. Jetons un coup d'œil au point de départ du code Dart de test unitaire : *le package de test Dart*.

4.8.2 Le package de test Dart

Le package de test Dart ne fait pas partie du SDK lui-même, il doit donc être installé en tant que dépendance tierce normale. Vous devriez déjà savoir comment faire cela.

Le code de test se trouve dans le dossier **test/**. Dans cet exemple (généré avec l'outil « Stagehand »), il existe une dépendance de développement. Il s'agit d'une dépendance requise uniquement pendant le développement et non à l'exécution :

```
dev_dependencies:  
  test:^1.0.0
```

Cela nous permet d'utiliser les bibliothèques fournies par le package de test pour écrire des tests unitaires.

Pour cela, nous allons dans un premier temps, vérifier que le package **test** est présent dans le fichier **pubspec.yaml**.

```
pubspec.yaml  
1  name: app48_2  
2  description: A sample command-line application.  
3  # version: 1.0.0  
4  # homepage: https://www.example.com  
5  
6  environment:  
7  |  sdk: '>=2.8.1 <3.0.0'  
8  
9  dependencies:  
10 |  path: ^1.7.0  
11  
12 dev_dependencies:  
13 |  pedantic: ^1.9.0  
14 |  test: ^1.14.4
```

Puis, nous devons générer les dépendances à l'aide de la commande **flutter pub get** dans une fenêtre de commande ou dans le terminal Visual Studio Code.

```
Microsoft Windows [version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.  
  
C:\tools\AppsDart\app48_2>flutter pub get  
Running "flutter pub get" in app48_2...  
2,3s  
C:\tools\AppsDart\app48_2>
```

Nous pouvons ensuite modifier les fichiers librairie(**lib/app48_2.dart**), principal(**app48_2.dart**) et test (**app48_2_test.dart**) respectivement dans les répertoires **lib/**, **bin/** et **test/**.

```
lib > lib_app48_2.dart > ...  
1  class Calculatrice {  
2    num additionDeuxNombres(num a, num b) {  
3      return a + b;  
4    }  
5  }
```

```
bin > app48_2.dart > ...  
1  import 'package:app48_2/lib_app48_2.dart';  
2  
3  void main(List<String> arguments) {  
4    Calculatrice calculatrice = new Calculatrice();  
5    print('La somme de 1 + 2 = ${calculatrice.additionDeuxNombres(1, 2)}!');  
6  }
```

```
test > app48_2_test.dart > main
  1 import 'package:test/test.dart';
  2 import 'package:app48_2/app48_2.dart';
  3
  4   Run | Debug
  5 void main() {
  6   Calculatrice calculatrice;
  7
  8   setUp(() {
  9     calculatrice = Calculatrice();
 10   });
 11
 12   test('Calculatrice addition de deux nombre calculate', () {
 13     expect(calculatrice.additionDeuxNombres(1, 2), 3);
 14   });
}
```

Vous pouvez maintenant effectuer le test unitaire, dans une fenêtre de commande ou dans le terminal Visual Studio Code, en exécutant la commande suivante de le répertoire du projet.

```
flutter pub run test test/app48_2_test.dart
```

Si le test est fructueux, vous obtenez la réponse ci-dessous :

```
C:\tools\AppsDart\app48_2>flutter pub run test test/app48_2_test.dart
Precompiling executable...
Precompiled test:test.
00:00 +0: Calculatrice addition de deux nombre calculate
00:00 +1: All tests passed!

C:\tools\AppsDart\app48_2>
```

Dans le cas contraire, la commande vous retourne les erreurs [E] :

```
C:\tools\AppsDart\app48_2>flutter pub run test test/app48_2_test.dart
00:00 +0: Calculatrice addition de deux nombre calculate
00:00 +0 -1: Calculatrice addition de deux nombre calculate [E]
  Expected: <4>
  Actual: <3>

  package:test_api          expect
  test\app48_2_test.dart 12:5  main.<fn>

00:00 +0 -1: Some tests failed.
pub finished with exit code 1

C:\tools\AppsDart\app48_2>
```

Vous pouvez également créer des groupes de tests, car vous pensez peut-être qu'un seul cas n'est pas suffisant pour tester efficacement une unité de code.

Supposons que nous changions notre suite de tests pour avoir un groupe de tests de somme, comme suit:

The screenshot shows a code editor with a Dart file named `app48_2_test.dart`. The code defines a `main` function that sets up a `Calculatrice` instance and runs a group of tests for addition. One test fails because it tries to add `null` to an integer, resulting in a `NoSuchMethodError`.

```
test > app48_2_test.dart > main
  3
    Run | Debug
  4 void main() {
  5   Calculatrice calculatrice;
  6
  7   setUp(() {
  8     calculatrice = Calculatrice();
  9   });
 10
 11   Run | Debug
 12   group("tests addition", () {
 13     Run | Debug
 14     test('Calculatrice addition de deux nombre', () {
 15       expect(calculatrice.additionDeuxNombres(1, 2), 3);
 16     });
 17     Run | Debug
 18     test('Calculatrice addition de null comme 0 et un nombre', () {
 19       expect(calculatrice.additionDeuxNombres(1, null), 1);
 20     });
 21   });
 22
 23
 24
 25
 26
 27
 28
 29
 30
 31
 32
 33
 34
 35
 36
 37
 38
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48
 49
 50
 51
 52
 53
 54
 55
 56
 57
 58
 59
 60
 61
 62
 63
 64
 65
 66
 67
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525
 526
 527
 528
 529
 530
 531
 532
 533
 534
 535
 536
 537
 538
 539
 540
 541
 542
 543
 544
 545
 546
 547
 548
 549
 550
 551
 552
 553
 554
 555
 556
 557
 558
 559
 559
 560
 561
 562
 563
 564
 565
 566
 567
 568
 569
 570
 571
 572
 573
 574
 575
 576
 577
 578
 579
 580
 581
 582
 583
 584
 585
 586
 587
 588
 589
 589
 590
 591
 592
 593
 594
 595
 596
 597
 598
 599
 599
 600
 601
 602
 603
 604
 605
 606
 607
 608
 609
 609
 610
 611
 612
 613
 614
 615
 616
 617
 618
 619
 619
 620
 621
 622
 623
 624
 625
 626
 627
 628
 629
 629
 630
 631
 632
 633
 634
 635
 636
 637
 638
 639
 639
 640
 641
 642
 643
 644
 645
 646
 647
 648
 649
 649
 650
 651
 652
 653
 654
 655
 656
 657
 658
 659
 659
 660
 661
 662
 663
 664
 665
 666
 667
 668
 669
 669
 670
 671
 672
 673
 674
 675
 676
 677
 678
 679
 679
 680
 681
 682
 683
 684
 685
 686
 687
 688
 689
 689
 690
 691
 692
 693
 694
 695
 696
 697
 698
 699
 699
 700
 701
 702
 703
 704
 705
 706
 707
 708
 709
 709
 710
 711
 712
 713
 714
 715
 716
 717
 718
 719
 719
 720
 721
 722
 723
 724
 725
 726
 727
 728
 729
 729
 730
 731
 732
 733
 734
 735
 736
 737
 738
 739
 739
 740
 741
 742
 743
 744
 745
 746
 747
 748
 749
 749
 750
 751
 752
 753
 754
 755
 756
 757
 758
 759
 759
 760
 761
 762
 763
 764
 765
 766
 767
 768
 769
 769
 770
 771
 772
 773
 774
 775
 776
 777
 778
 779
 779
 780
 781
 782
 783
 784
 785
 786
 787
 788
 789
 789
 790
 791
 792
 793
 794
 795
 796
 797
 798
 799
 799
 800
 801
 802
 803
 804
 805
 806
 807
 808
 809
 809
 810
 811
 812
 813
 814
 815
 816
 817
 818
 819
 819
 820
 821
 822
 823
 824
 825
 826
 827
 828
 829
 829
 830
 831
 832
 833
 834
 835
 836
 837
 838
 839
 839
 840
 841
 842
 843
 844
 845
 846
 847
 848
 849
 849
 850
 851
 852
 853
 854
 855
 856
 857
 858
 859
 859
 860
 861
 862
 863
 864
 865
 866
 867
 868
 869
 869
 870
 871
 872
 873
 874
 875
 876
 877
 878
 879
 879
 880
 881
 882
 883
 884
 885
 886
 887
 888
 889
 889
 890
 891
 892
 893
 894
 895
 896
 897
 898
 899
 899
 900
 901
 902
 903
 904
 905
 906
 907
 908
 909
 909
 910
 911
 912
 913
 914
 915
 916
 917
 918
 919
 919
 920
 921
 922
 923
 924
 925
 926
 927
 928
 929
 929
 930
 931
 932
 933
 934
 935
 936
 937
 938
 939
 939
 940
 941
 942
 943
 944
 945
 946
 947
 948
 949
 949
 950
 951
 952
 953
 954
 955
 956
 957
 958
 959
 959
 960
 961
 962
 963
 964
 965
 966
 967
 968
 969
 969
 970
 971
 972
 973
 974
 975
 976
 977
 978
 979
 979
 980
 981
 982
 983
 984
 985
 986
 987
 988
 989
 989
 990
 991
 992
 993
 994
 995
 996
 997
 998
 999
 999
 1000
 1001
 1002
 1003
 1004
 1005
 1006
 1007
 1008
 1009
 1009
 1010
 1011
 1012
 1013
 1014
 1015
 1016
 1017
 1018
 1019
 1019
 1020
 1021
 1022
 1023
 1024
 1025
 1026
 1027
 1028
 1029
 1029
 1030
 1031
 1032
 1033
 1034
 1035
 1036
 1037
 1038
 1039
 1039
 1040
 1041
 1042
 1043
 1044
 1045
 1046
 1047
 1048
 1049
 1049
 1050
 1051
 1052
 1053
 1054
 1055
 1056
 1057
 1058
 1059
 1059
 1060
 1061
 1062
 1063
 1064
 1065
 1066
 1067
 1068
 1069
 1069
 1070
 1071
 1072
 1073
 1074
 1075
 1076
 1077
 1078
 1079
 1079
 1080
 1081
 1082
 1083
 1084
 1085
 1086
 1087
 1088
 1089
 1089
 1090
 1091
 1092
 1093
 1094
 1095
 1096
 1097
 1098
 1099
 1099
 1100
 1101
 1102
 1103
 1104
 1105
 1106
 1107
 1108
 1109
 1109
 1110
 1111
 1112
 1113
 1114
 1115
 1116
 1117
 1118
 1119
 1119
 1120
 1121
 1122
 1123
 1124
 1125
 1126
 1127
 1128
 1129
 1129
 1130
 1131
 1132
 1133
 1134
 1135
 1136
 1137
 1138
 1139
 1139
 1140
 1141
 1142
 1143
 1144
 1145
 1146
 1147
 1148
 1148
 1149
 1150
 1151
 1152
 1153
 1154
 1155
 1156
 1157
 1158
 1159
 1159
 1160
 1161
 1162
 1163
 1164
 1165
 1166
 1167
 1168
 1169
 1169
 1170
 1171
 1172
 1173
 1174
 1175
 1176
 1177
 1178
 1178
 1179
 1180
 1181
 1182
 1183
 1184
 1185
 1186
 1187
 1187
 1188
 1189
 1190
 1191
 1192
 1193
 1194
 1195
 1196
 1197
 1197
 1198
 1199
 1199
 1200
 1201
 1202
 1203
 1204
 1205
 1206
 1207
 1208
 1209
 1209
 1210
 1211
 1212
 1213
 1214
 1215
 1216
 1217
 1218
 1219
 1219
 1220
 1221
 1222
 1223
 1224
 1225
 1226
 1227
 1228
 1229
 1229
 1230
 1231
 1232
 1233
 1234
 1235
 1236
 1237
 1238
 1239
 1239
 1240
 1241
 1242
 1243
 1244
 1245
 1246
 1247
 1248
 1248
 1249
 1250
 1251
 1252
 1253
 1254
 1255
 1256
 1257
 1258
 1259
 1259
 1260
 1261
 1262
 1263
 1264
 1265
 1266
 1267
 1268
 1269
 1269
 1270
 1271
 1272
 1273
 1274
 1275
 1276
 1277
 1278
 1278
 1279
 1280
 1281
 1282
 1283
 1284
 1285
 1286
 1287
 1288
 1288
 1289
 1290
 1291
 1292
 1293
 1294
 1295
 1296
 1297
 1297
 1298
 1299
 1299
 1300
 1301
 1302
 1303
 1304
 1305
 1306
 1307
 1308
 1309
 1309
 1310
 1311
 1312
 1313
 1314
 1315
 1316
 1317
 1318
 1319
 1319
 1320
 1321
 1322
 1323
 1324
 1325
 1326
 1327
 1328
 1329
 1329
 1330
 1331
 1332
 1333
 1334
 1335
 1336
 1337
 1338
 1338
 1339
 1340
 1341
 1342
 1343
 1344
 1345
 1346
 1347
 1348
 1348
 1349
 1350
 1351
 1352
 1353
 1354
 1355
 1356
 1357
 1358
 1359
 1359
 1360
 1361
 1362
 1363
 1364
 1365
 1366
 1367
 1368
 1368
 1369
 1370
 1371
 1372
 1373
 1374
 1375
 1376
 1377
 1378
 1378
 1379
 1380
 1381
 1382
 1383
 1384
 1385
 1386
 1387
 1387
 1388
 1389
 1389
 1390
 1391
 1392
 1393
 1394
 1395
 1396
 1397
 1398
 1398
 1399
 1400
 1401
 1402
 1403
 1404
 1405
 1406
 1407
 1408
 1409
 1409
 1410
 1411
 1412
 1413
 1414
 1415
 1416
 1417
 1418
 1419
 1419
 1420
 1421
 1422
 1423
 1424
 1425
 1426
 1427
 1428
 1429
 1429
 1430
 1431
 1432
 1433
 1434
 1435
 1436
 1437
 1438
 1438
 1439
 1440
 1441
 1442
 1443
 1444
 1445
 1446
 1447
 1448
 1448
 1449
 1450
 1451
 1452
 1453
 1454
 1455
 1456
 1457
 1458
 1459
 1459
 1460
 1461
 1462
 1463
 1464
 1465
 1466
 1467
 1468
 1468
 1469
 1470
 1471
 1472
 1473
 1474
 1475
 1476
 1477
 1478
 1478
 1479
 1480
 1481
 1482
 1483
 1484
 1485
 1486
 1487
 1488
 1488
 1489
 1490
 1491
 1492
 1493
 1494
 1495
 1496
 1497
 1498
 1498
 1499
 1500
 1501
 1502
 1503
 1504
 1505
 1506
 1507
 1508
 1509
 1509
 1510
 1511
 1512
 1513
 1514
 1515
 1516
 1517
 1518
 1519
 1519
 1520
 1521
 1522
 1523
 1524
 1525
 1526
 1527
 1528
 1529
 1529
 1530
 1531
 1532
 1533
 1534
 1535
 1536
 1537
 1538
 1538
 1539
 1540
 1541
 1542
 1543
 1544
 1545
 1546
 1547
 1548
 1548
 1549
 1550
 1551
 1552
 1553
 1554
 1555
 1556
 1557
 1558
 1559
 1559
 1560
 1561
 1562
 1563
 1564
 1565
 1566
 1567
 1568
 1568
 1569
 1570
 1571
 1572
 1573
 1574
 1575
 1576
 1577
 1578
 1578
 1579
 1580
 1581
 1582
 1583
 1584
 1585
 1586
 1587
 1588
 1588
 1589
 1590
 1591
 1592
 1593
 1594
 1595
 1596
 1597
 1598
 1598
 1599
 1600
 1601
 1602
 1603
 1604
 1605
 1606
 1607
 1608
 1609
 1609
 1610
 1611
 1612
 1613
 1614
 1615
 1616
 1617
 1618
 1619
 1619
 1620
 1621
 1622
 1623
 1624
 1625
 1626
 1627
 1628
 1629
 1629
 1630
 1631
 1632
 1633
 1634
 1635
 1636
 1637
 1638
 1638
 1639
 1640
 1641
 1642
 1643
 1644
 1645
 1646
 1647
 1648
 1648
 1649
 1650
 1651
 1652
 1653
 1654
 1655
 1656
 1657
 1658
 1659
 1659
 1660
 1661
 1662
 1663
 1664
 1665
 1666
 1667
 1668
 1668
 1669
 1670
 1671
 1672
 1673
 1674
 1675
 1676
 1677
 1678
 1678
 1679
 1680
 1681
 1682
 1683
 1684
 1685
 1686
 1687
 1688
 1688
 1689
 1690
 1691
 1692
 1693
 1694
 1695
 1696
 1697
 1698
 1698
 1699
 1700
 1701
 1702
 1703
 1704
 1705
 1706
 1707
 1708
 1709
 1709
 1710
 1711
 1712
 1713
 1714
 1715
 1716
 1717
 1718
 1719
 1719
 1720
 1721
 1722
 1723
 1724
 1725
 1726
 1727
 1728
 1729
 1729
 1730
 1731
 1732
 1733
 1734
 1735
 1736
 1737
 1738
 1738
 1739
 1740
 1741
 1742
 1743
 1744
 1745
 1746
 1747
 1748
 1748
 1749
 1750
 1751
 1752
 1753
 1754
 1755
 1756
 1757
 1758
 1759
 1759
 1760
 1761
 1762
 1763
 1764
 1765
 1766
 1767
 1768
 1768
 1769
 1770
 1771
 1772
 1773
 1774
 1775
 1776
 1777
 1778
 1779
 1779
 1780
 1781
 1782
 1783
 1784
 1785
 1786
 1787
 1788
 1788
 1789
 1790
 1791
 1792
 1793
 1794
 1795
 1796
 1797
 1798
 1798
 1799
 1800
 1801
 1802
 1803
 1804
 1805
 1806
 1807
 1808
 1809
 1809
 1810
 1811
 1812
 1813
 1814
 1815
 1816
 1817
 1818
 1819
 1819
 1820
 1821
 1822
 1823
 1824
 1825
 1826
 1827
 1828
 1829
 1829
 1830
 1831
 1832
 1833
 1834
 1835
 1836
 1837
 1838
 1838
 1839
 1840
 1841
 1842
 1843
 1844
 1845
 1846
 1847
 1848
 1848
 1849
 1850
 1851
 1852
 1853
 1854
 1855
 1856
 1857
 1858
 1859
 1859
 1860
 1861
 1862
 1863
 1864
 1865
 1866
 1867
 1868
 1868

```

5 Flutter

5.1 Introduction à Flutter

Vous allez apprendre l'histoire du framework Flutter, comment et pourquoi il a été créé et son évolution jusqu'à présent. Vous apprendrez comment sa communauté y contribue, et comment et pourquoi elle s'est développée rapidement ces derniers mois.

Il vous sera présenté les principales fonctionnalités de Flutter, avec de brèves comparaisons avec d'autres frameworks. Vous verrez également comment créer un projet Flutter de base. Pour ce faire, nous aurons besoin d'une machine appropriée configurée avec Flutter et ses différents prérequis (voir annexe 6.3).

5.2 Comparaisons avec d'autres frameworks de développement d'applications mobiles

5.2.1 introduction

Bien qu'il soit relativement nouveau, Flutter a connu beaucoup d'expérimentation et d'évolution au fil des ans.

Il s'appelait Sky, lors de sa première apparition au *Dart Developer Summit 2015* présenté par Eric Seidel. Il a été présenté comme l'évolution de certaines expériences précédentes de Google pour créer quelque chose de mieux pour les mobiles, en termes de développement et d'expérience utilisateur, avec pour objectif principal un rendu haute performance. Présenté sous le nom de Flutter en 2016, et avec sa première version alpha en mai 2017, il se construisait déjà pour les systèmes iOS et Android. Ensuite, il a commencé à mûrir et l'adoption communautaire a commencé à se développer. Il a évolué des commentaires de la communauté à sa première version stable fin 2018.

Il existe de nombreux frameworks de développement mobile qui visent un objectif commun : créer des applications mobiles natives pour Android et iOS avec une seule base de code. Certains de ces cadres sont largement adoptés par la communauté et fournissent des solutions similaires aux problèmes qu'ils prétendent résoudre.

Sachant cela, nous pouvons nous demander ce qui suit :

- Pourquoi Flutter a-t-il été créé ?
- En avons-nous vraiment besoin ?
- En quoi est-ce mieux que les concurrents ?

Voyons comment fonctionne Flutter et répondons à certaines de ces questions avant de mettre la main dessus.

5.2.2 Les problèmes que Flutter veut résoudre

Depuis le début du framework Flutter, il était destiné à offrir une meilleure expérience à l'utilisateur grâce à une exécution haute performance, mais ce n'est pas la seule promesse de Flutter. L'expérience de développement a également été axée sur la résolution de certains des problèmes du développement mobile sur plusieurs plateformes :

Cycles de développement longs / plus coûteux : pour pouvoir faire face à la demande du marché, vous devez choisir de créer une plateforme unique ou de créer plusieurs équipes. Cela a des conséquences en termes de coût, de délais multiples et de capacités différentes des frameworks natifs.

Plusieurs langages à apprendre : si un développeur souhaite développer pour plusieurs plateformes, il doit apprendre à travailler avec un système d'exploitation et un langage de programmation, et plus tard, la même chose sur un autre système d'exploitation et un autre langage de programmation. Cela a certainement un impact sur la productivité du développeur.

Temps de construction/compilation long : certains développeurs ont peut-être déjà compris comment le temps de compilation peut avoir un impact sur la productivité. Sous Android, par exemple, certains développeurs connaissent des temps d'attentes importants pendant la construction de paquets apk après seulement quelques minutes de codage (cela évolue, et c'est beaucoup mieux maintenant, mais cela causait déjà beaucoup de douleur).

Effets secondaires des solutions multi-plateformes existantes : vous adoptez un univers multi-plateforme existant pour tenter de contourner les problèmes précédents, mais cela pourrait avoir un impact sur les performances, la conception ou l'expérience de l'utilisateur.

Voyons maintenant comment Flutter résout ces problèmes.

5.2.3 Différences entre les frameworks existants

Il existe un grand nombre de frameworks et de technologies de haute qualité et bien acceptés. Certains d'entre eux sont : Xamarin, React Native, Native Script, Ionic, Cordova...

Ainsi, vous pourriez penser qu'il est difficile pour un nouveau framework de trouver sa place sur un terrain déjà plein, mais ce n'est pas le cas. Flutter présente des avantages qui lui permettent de se faire une place, pas nécessairement en surmontant les autres frameworks, mais en étant déjà au moins au même niveau que les frameworks natifs :

- Hautes performances
- Contrôle total de l'interface utilisateur
- Langage Dart
- Support par Google
- Open source framework
- Ressources pour les développeurs

Examinons chacun de ces éléments plus en détail.

5.2.3.1 Hautes performances

À l'heure actuelle, il est difficile de dire que les performances de Flutter sont toujours meilleures que celles de tous les autres frameworks, mais il est sûr que nous pouvons dire qu'elle vise à l'être. Par exemple, sa couche de rendu a été développée en gardant à l'esprit une fréquence d'images élevée. Comme nous le verrons dans la section de rendu Flutter, certains des frameworks existants reposent sur le rendu JavaScript et HTML, ce qui peut entraîner des surcoûts de performances car tout est dessiné dans une vue Web (un composant visuel comme un navigateur Web). Certains utilisent des widgets OEM (Original Equipment Manufacturer), mais s'appuient sur un pont pour demander à l'API du système d'exploitation de rendre les composants, ce qui crée un goulet d'étranglement dans l'application, car elle nécessite une étape supplémentaire pour rendre l'interface utilisateur (UI).

Quelques points qui rendent les performances de Flutter excellentes :

Flutter gère le pixel : Flutter dessine l'application pixel par pixel, en interagissant directement avec le moteur graphique **Skia**.

Pas de couches supplémentaires ou d'appels à l'API de OS : comme Flutter possède le contrôle et l'affichage du rendu de l'application, il n'a pas besoin d'appels supplémentaires pour utiliser les widgets OEM.

Flutter est compilé en code natif : Flutter utilise le compilateur Dart AOT pour produire du code natif. Cela signifie qu'il n'y a pas de surcharge dans la configuration d'un environnement pour interpréter le code Dart à la volée, et il fonctionne comme une application native, démarrant plus rapidement que les Framework qui nécessitent une sorte d'interpréteur.

5.2.3.2 Contrôle total de l'interface utilisateur

Le framework Flutter gère complètement toute l'interface utilisateur, affichant les composants visuels directement sur le « *canvas* » de l'écran. Flutter ne nécessitant rien de plus que le « *canvas* » de la plate-forme afin qu'il ne soit pas limité par des règles et des conventions. La plupart du temps, les frameworks ne font que reproduire ce que propose la plateforme d'une autre manière. Par exemple, d'autres frameworks multi plateformes basés sur la vue Web reproduisent des composants visuels à l'aide d'éléments HTML avec un style CSS. D'autres frameworks émulent la création des composants visuels et les transmettent à la plate-forme de l'appareil, qui rendra les widgets OEM comme une application développée en natif. Nous ne parlons pas ici de performances, alors qu'est-ce que Flutter offre d'autre en n'utilisant pas les widgets OEM et en faisant le travail tout seul ?

Gérer tous les pixels de l'appareil : les cadres limités par les widgets OEM reproduiront au maximum ce qu'une application développée en natif ferait, car ils n'utilisent que les composants disponibles de la plate-forme. D'autre part, les frameworks basés sur les technologies Web peuvent reproduire plus que des composants spécifiques à la plate-forme, mais peuvent également être limités par le moteur Web mobile disponible sur l'appareil. En obtenant le contrôle du rendu de l'interface utilisateur, Flutter permet au développeur de créer l'interface utilisateur à sa manière en exposant une API Widgets extensible et riche, qui fournit des outils pouvant être utilisés pour créer une interface utilisateur unique sans inconvénients en termes de performances et sans limites de la conception.

Kits d'interface utilisateur dédié à une plate-forme : en n'utilisant pas de widgets OEM, Flutter pourrait casser la conception de la plate-forme, mais ce n'est pas le cas. Flutter est équipé de packages qui fournissent des widgets de conception liés à la plate-forme, un jeu Android pour Android (normal 😊) et un jeu Cupertino sous iOS.

Exigences de conception d'interface utilisateur : Flutter fournit une API propre et robuste avec la possibilité de reproduire des mises en page fidèles aux exigences de conception. Contrairement aux frameworks basés sur le Web qui reposent sur des règles de mise en page CSS qui peuvent être volumineuses, compliquées et même contradictoires, Flutter simplifie cela en ajoutant des règles sémantiques qui peuvent être utilisées pour créer des mises en page complexes mais efficaces.

Une apparence et une sensation de fluidités : en plus des kits de widgets natifs, Flutter cherche à fournir une expérience de plate-forme native où l'application s'exécute, de sorte que les polices, les gestes et les interactions sont implémentés d'une manière spécifique à la plate-forme, apportant une sensation naturelle à l'utilisateur, comme une application native.

5.2.3.3 Utilisation du langage Dart

Depuis sa création, l'un des principaux objectifs de Flutter était d'être une alternative haute performance aux frameworks multiplateformes existants. Améliorer significativement l'expérience du développeur mobile était l'un des points cruciaux du projet.

Dans cet esprit, Flutter avait besoin d'un langage de programmation qui lui permette d'atteindre ces objectifs, et Dart semble correspondre parfaitement à cela pour les raisons suivantes :

Compilation Dart AOT (Ahead Of Time) et JIT (Just In Time) : Dart est suffisamment flexible pour fournir différentes manières d'exécuter un code. Flutter utilise Dart AOT (Ahead Of Time) avec une idée de performances à l'esprit lors de la compilation d'une version finale de l'application, et il utilise JIT avec une compilation de code en moins d'une seconde au moment du développement, visant des flux de travail rapides et des changements de code.

Rappel : Les compilations Dart Just in time (JIT) et Ahead of time (AOT) sont introduites lors de la phase de compilation. Dans AOT, le code est compilé avant de s'exécuter. Dans JIT, le code est compilé lors de l'exécution.

Haute performance : en raison de la prise en charge de la compilation AOT par Dart, Flutter ne nécessite pas de lien lent entre les environnements (par exemple, non natif à natif), ce qui permet aux applications Flutter de démarrer beaucoup plus rapidement. En outre, Flutter utilise un flux de style fonctionnel avec des objets de courte durée, ce qui signifie beaucoup d'allocations de courte durée. Le « garbage collector » Dart fonctionne sans verrous, ce qui permet une allocation rapide.

Apprentissage facile : Dart est un langage flexible, robuste, moderne et avancé. Bien qu'il soit encore en évolution, le langage est un langage orienté objet bien défini avec des fonctionnalités familières aux langages dynamiques et statiques. Il dispose d'une communauté active et d'une documentation bien structurée.

Interface utilisateur déclarative : dans Flutter, nous utilisons un style déclaratif pour mettre en page les widgets, ce qui signifie que les widgets sont immuables. Pour changer l'interface utilisateur, un widget déclenche une reconstruction sur lui-même et sur son sous-arbre. Dans le style impératif opposé (le plus courant), nous pouvons modifier les propriétés des composants spécifiques après leur création.

Syntaxe Dart à la mise en page : Différent de nombreux frameworks qui ont une syntaxe distincte pour la mise en page et le code, dans Flutter, la mise en page est créée à l'aide du code Dart, visant une plus grande flexibilité et une facilité de création d'un environnement de développement, avec des outils de débogage des performances de rendu de mise en page.

5.2.3.4 Support par Google

Flutter est un tout nouveau framework, ce qui signifie qu'il n'est pas encore implémenté dans une grande partie du marché du développement mobile, mais cela change et les perspectives pour les prochaines années sont très positives.

Supporté par Google, le framework dispose de tous les outils nécessaires pour réussir dans la communauté, la présence à de grands événements tels que *Google IO* et des investissements dans l'amélioration continue de la base de code. Du lancement de la troisième version bêta à *Google IO 2018* à la première version stable lancée lors du *Flutter Live Event* fin 2018, sa croissance est évidente :

- ⊕ Plus de 200 millions d'utilisateurs d'applications Flutter.
- ⊕ Plus de 3000 applications Flutter sur le Play Store.
- ⊕ Plus de 250 000 nouveaux développeurs.

De plus, ce n'est plus un secret pour personne que Google travaille sur son nouveau système d'exploitation Fuchsia en remplacement du système d'exploitation Android. Une chose à laquelle il

faut faire attention est que OS Fuchsia est un système Google universel pour fonctionner sur plus que les téléphones mobiles, et cela affecte directement l'adoption de Flutter.

En effet, Flutter sera la première méthode de développement d'applications mobiles pour le nouveau système d'exploitation Fuchsia, et non seulement cela, l'interface utilisateur du système est en cours de développement avec lui. Le système ciblant plus d'appareils que de simples smartphones, comme cela semble être le cas, Flutter aura certainement beaucoup d'améliorations.

La croissance de l'adoption du framework est directement liée au nouveau système d'exploitation Fuchsia. À l'approche du lancement, il est important pour Google de disposer d'applications mobiles ciblant le nouveau système. Par exemple, Google a annoncé que les applications Android seront compatibles avec le nouveau système d'exploitation, ce qui facilitera considérablement la transition et l'adoption de Flutter.

5.2.3.5 Framework open source

Avoir une grande entreprise comme Google derrière est fondamental pour le framework Flutter (voir React, par exemple, qui est maintenu par Facebook).

De plus, le soutien de la communauté devient encore plus important à mesure qu'il devient plus populaire. En étant open source, la communauté et Google peuvent travailler ensemble pour:

- Aider à la résolution de bogues et à la documentation via la collaboration de code
- Créer un nouveau contenu éducatif sur le framework
- Documentation et utilisation de support
- Prendre des décisions d'amélioration basées sur de vrais commentaires

L'amélioration de l'expérience des développeurs est l'un des principaux objectifs du framework. Par conséquent, en plus d'être proche de la communauté, le framework fournit d'excellents outils et ressources pour les développeurs.

5.2.3.6 Outils pour les développeurs

L'accent mis sur les développeurs dans le framework Flutter va de la documentation et des ressources d'apprentissage à la fourniture d'outils pour aider à la productivité :

Documentation et ressources d'apprentissage : les sites Web Flutter sont riches pour les développeurs provenant d'autres plates-formes, y compris de nombreux exemples et cas d'utilisation, pour exemple, les fameux Google Codelabs (<https://codelabs.developers.google.com/?cat=Flutter>).

Outils en ligne de commande et intégration IDE : les outils Dart qui aident à analyser, exécuter et gérer les dépendances font également partie de Flutter. En plus de cela, Flutter propose également des commandes pour aider au débogage, au déploiement, à l'inspection du rendu de disposition et à l'intégration avec les IDE via les plugins Dart.

Voici une liste des différentes commandes :

```

C:\Users\didier.jugehubert.IMTLD>flutter --help
Manage your Flutter app development.

Common commands:
  flutter create <output directory>
    Create a new Flutter project in the specified directory.

  flutter run [options]
    Run your Flutter application on an attached device or in an emulator.

Usage: flutter <command> [arguments]

Global options:
  -h, --help                  Print this usage information.
  -v, --verbose                Noisy logging, including all shell commands
                                executed.
  -d, --device-id              If used with --help, shows hidden options.
                                Target device id or name <prefixes allowed>.
  --version                   Reports the version of this tool.
  --suppress-analytics        Suppress analytics reporting when this command runs.

  --packages                  Path to your ".packages" file.
                                (required, since the current directory does not
                                contain a ".packages" file)

Available commands:
  analyze                      Analyze the project's Dart code.
  assemble                     Assemble and build flutter resources.
  attach                       Attach to a running application.
  bash-completion              Output command line shell completion setup scripts.
  build                        Flutter build commands.
  channel                      List or switch flutter channels.
  clean                         Delete the build/ and .dart_tool/ directories.
  config                        Configure Flutter settings.
  create                        Create a new Flutter project.
  devices                       List all connected devices.
  doctor                        Show information about the installed tooling.
  downgrade                    Downgrade Flutter to the last active version for the current
                                channel.
  drive                         Runs Flutter Driver tests for the current project.
  emulators                     List, launch and create emulators.
  format                        Format one or more dart files.
  install                       Install a Flutter app on an attached device.
  logs                          Show log output for running Flutter apps.
  precache                      Populates the Flutter tool's cache of binary artifacts.
  pub                            Commands for managing Flutter packages.
  run                            Run your Flutter app on an attached device.
  screenshot                    Take a screenshot from a connected device.
  symbolize                     Symbolize a stack trace from an AOT compiled flutter
                                application.
  test                           Run Flutter unit tests for the current project.
  upgrade                       Upgrade your copy of Flutter.

Run "flutter help <command>" for more information about a command.
Run "flutter help -v" for verbose help output, including less commonly used
options.

C:\Users\didier.jugehubert.IMTLD>

```

Mise en œuvre simple : Flutter est livré avec l'outil « *flutter doctor* », qui est un outil ligne de commande qui guide le développeur à travers la configuration du système en indiquant ce qui est nécessaire pour configurer un environnement Flutter.

```

PS C:\Users\didier.jugehubert.IMTLD> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel stable, 1.20.2, on Microsoft Windows [version 6.1.7601], locale fr-FR)
[!] Android toolchain - develop for Android devices (Android SDK version 30.0.2)
[!] Android Studio (version 4.0)
[!] VS Code, 64-bit edition (version 1.48.0)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
PS C:\Users\didier.jugehubert.IMTLD>

```

La commande « *flutter doctor* » identifie également les appareils connectés et s'il existe des mises à jour disponibles.

Rechargement « à chaud » : c'est la fonctionnalité qui a été au centre des présentations sur le framework. En combinant les capacités du langage Dart (comme la compilation JIT) et la puissance de Flutter, il est possible pour le développeur de voir instantanément les modifications de conception apportées au code dans le simulateur ou l'appareil connecté. Dans Flutter, il n'y a pas d'outil spécifique pour l'aperçu de la mise en page.

5.2.4 Compilation Flutter (Dart)

La façon dont une application est construite est fondamentale pour ses performances sur la plate-forme cible. Il s'agit d'une étape importante concernant les performances. Même si vous n'avez pas nécessairement besoin de le savoir pour chaque type d'application, savoir comment l'application est créée vous aide à comprendre et à mesurer les améliorations possibles.

Comme nous l'avons déjà souligné, Flutter s'appuie sur la compilation AOT de Dart pour le mode release et la compilation JIT de Dart pour le mode développement/débogage. Dart est l'un des très rares langages capables d'être compilés à la fois en AOT et en JIT, et pour Flutter, c'est génial.

5.2.4.1 Compilation de développement

Pendant le développement, Flutter utilise la compilation JIT en mode développement. Cela permet des fonctionnalités de développement importantes telles que le rechargement à chaud.

En raison de la puissance du compilateur de Dart, les interactions entre le code et le simulateur/périphérique sont très rapides et les informations de débogage aident les développeurs à entrer dans le code source.

5.2.4.2 Compilation de publication

En mode version, les informations de débogage ne sont pas nécessaires et l'accent est mis sur les performances. Flutter utilise une technique commune aux moteurs de jeu. En utilisant le mode AOT, le code Dart est compilé en code natif, et l'application charge la bibliothèque Flutter et lui délègue le rendu, l'entrée et la gestion des événements à l'aide du moteur Skia.

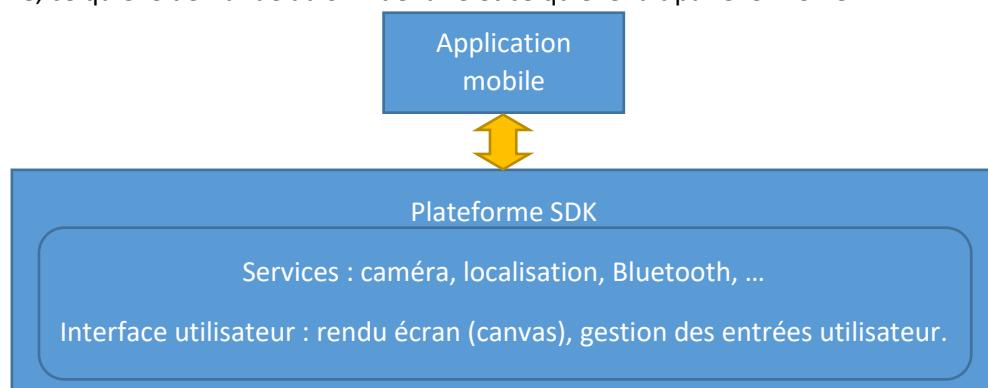
5.2.4.3 Plates-formes prises en charge

À présent, Flutter prend en charge les appareils ARM Android fonctionnant au moins sur la version Jelly Bean 4.1.x et les appareils iOS à partir de l'iPhone 4S ou plus récent. Bien sûr, les applications Flutter peuvent normalement être exécutées sur des simulateurs.

5.2.5 Rendu Flutter

5.2.5.1 Pour les applications mobiles

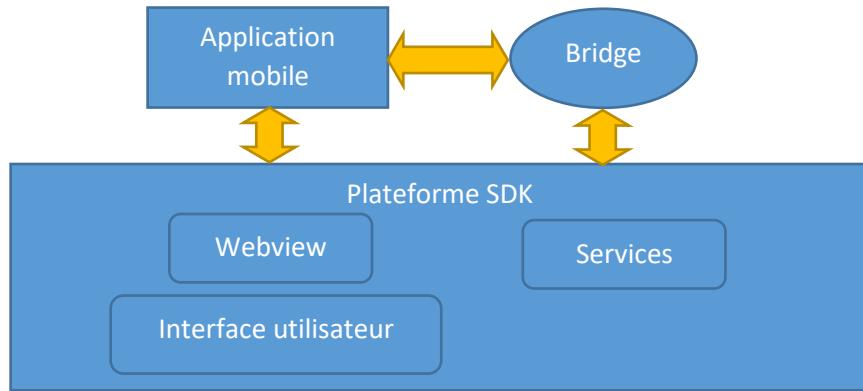
L'un des principaux aspects qui rend Flutter unique est la façon dont il dessine les composants visuels sur l'écran. La grande différence réside dans la manière dont l'application communique avec le SDK de la plate-forme, ce qu'elle demande au SDK de faire et ce qu'elle fait par elle-même :



Le SDK de la plate-forme peut être considéré comme l'interface entre les applications, le système d'exploitation et les services. Chaque système fournit son propre SDK avec ses propres capacités et est basé sur un langage de programmation (c'est-à-dire Kotlin/Java pour le SDK Android et Swift/Objective C pour le SDK iOS).

5.2.5.2 Pour les applications mobiles basées sur les technologies WEB

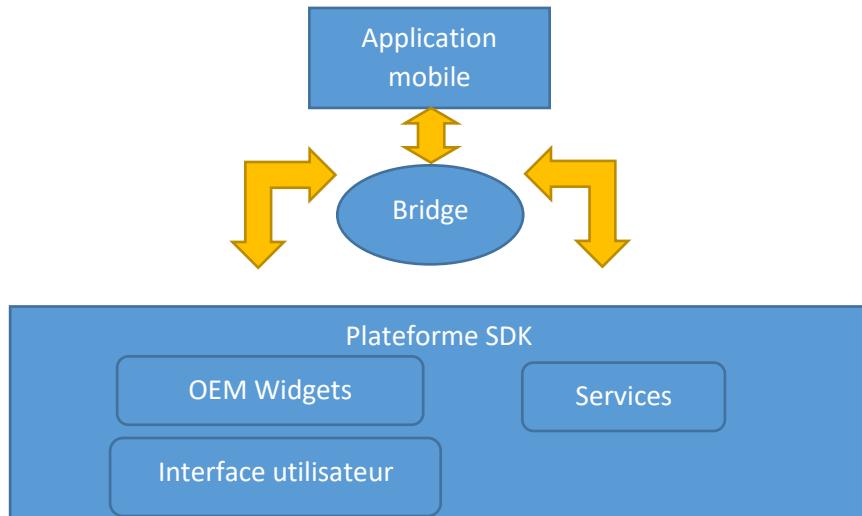
Nous avons déjà vu des frameworks qui utilisent des vues Web pour reproduire une interface utilisateur en combinant HTML et CSS. En termes d'utilisation de la plateforme, cela ressemblerait à ceci :



L'application ne sait pas comment le rendu est effectué par la plateforme; la seule chose dont elle a besoin est le widget de visualisation Web sur lequel elle enverra le code HTML et CSS.

5.2.5.3 Pour les applications mobiles via Framework et widgets OEM

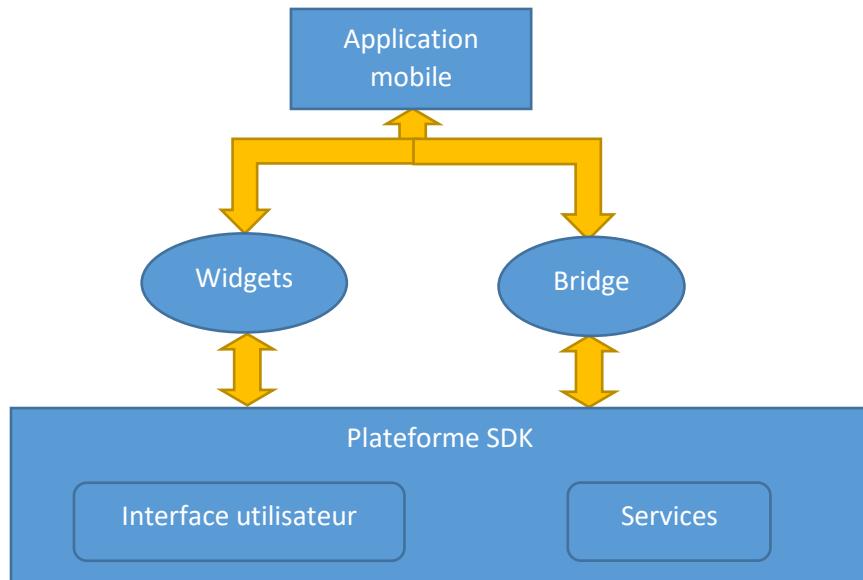
Une autre façon de rendre les widgets consiste à ajouter une couche au-dessus des widgets de la plate-forme, mais sans changer la façon dont le système affichera efficacement les composants visuels :



Dans ce mode de rendu, le travail est effectué par le SDK comme une application native normale, mais avant lui, la mise en page est définie par une étape supplémentaire dans le langage du framework. Chaque changement dans l'interface utilisateur provoque une communication entre le code de l'application et le code natif chargé d'appeler le SDK de la plate-forme, fonctionnant comme un intermédiaire. Comme la technique précédente, cela peut entraîner une petite surcharge pour l'application, peut-être un peu plus grande que la précédente.

5.2.5.4 Pour les applications mobiles via Framework uniquement

Flutter choisit de faire tout le travail « dur » par lui-même. La seule chose dont il a besoin du SDK de la plateforme est l'accès aux API de services et à un « canvas » sur lequel dessiner l'interface utilisateur :



Flutter déplace les widgets et le rendu vers l'application, d'où il obtient la personnalisation et l'extensibilité. Grâce à un « canvas », il peut dessiner n'importe quoi et accéder aux événements pour gérer les entrées et les gestes de l'utilisateur par lui-même. Le pont dans Flutter se fait par des canaux sur la plateforme.

5.2.6 Introduction aux widgets

Comprendre les widgets Flutter est essentiel si vous souhaitez travailler avec eux. Vous savez que Flutter prend le contrôle du rendu et le fait en gardant à l'esprit l'extensibilité et la personnalisation, dans le but d'ajouter de la puissance aux mains du développeur. Voyons comment Flutter applique l'idée des widgets dans le développement d'applications pour créer des interfaces utilisateur.

Les widgets Flutter sont partout dans une application. Peut-être que tout n'est pas un widget, mais presque tout l'est. Même l'application est un widget dans Flutter, et c'est pourquoi ce concept est si important.

Un widget représente une partie de l'interface utilisateur, mais cela ne signifie pas que ce n'est que quelque chose qui est visible. Il peut s'agir de l'un des éléments suivants :

- ✚ Un élément visuel / structurel qui est un élément de base, tel que les widgets Bouton ou Texte
- ✚ Un élément spécifique à la mise en page qui peut définir la position, les marges ou le remplissage, tel que le widget Remplissage
- ✚ Un élément de style qui peut aider à coloriser et à thématiser un élément visuel / structurel, tel que le widget *Theme*
- ✚ Un élément d'interaction qui aide à répondre aux interactions de différentes manières, tel que le widget *GestureDetector*.

Les widgets sont les éléments de base d'une interface. Pour créer une interface utilisateur correctement, flutter organise les widgets dans une arborescence de widgets.

Les widgets peuvent être assimilés comme la représentation visuelle (mais pas seulement) de parties de l'application. De nombreux widgets sont assemblés pour composer l'interface utilisateur d'une

application. Imaginez-le comme un puzzle dans lequel vous définissez les pièces. L'intention des widgets est de permettre à votre application d'être modulaire, évolutive et expressive, avec le moins de code et sans imposer de limitations.

Les principales caractéristiques de l'interface utilisateur des widgets dans flutter sont la « composabilité » et « l'immuabilité ».

5.2.6.1 Composabilité

Flutter choisit la composition plutôt que l'héritage, dans le but de garder chaque widget simple et avec un objectif bien défini. La flexibilité, qui est l'un des objectifs du framework, permet au développeur de faire de nombreuses combinaisons pour obtenir des résultats.

5.2.6.2 Immuabilité

Flutter est basé sur le style réactif de programmation, où les instances de widget sont de courte durée et changent leurs descriptions (visuellement ou non) en fonction des changements de configuration, de sorte qu'il réagit aux changements et propage ces changements à ses widgets de composition. Un widget flutter peut avoir un état associé, et lorsque l'état associé change, il peut être reconstruit pour correspondre à la représentation.

5.2.6.3 Arborescence des widgets

L'arborescence des widgets est la représentation logique de tous les widgets de l'interface utilisateur. Elle est calculée lors de la mise en page (mesures et informations structurelles) et utilisée pendant le rendu (image à écran) et les tests d'entrées (interactions tactiles), ce que Flutter fait le mieux. En utilisant de nombreux algorithmes d'optimisation, il essaie de manipuler le moins possible l'arborescence, réduisant la quantité totale de travail consacrée au rendu, visant une plus grande efficacité. Les widgets sont représentés dans l'arborescence sous forme de nœuds auxquels il peut être associé un état. Chaque modification de cet état entraîne la reconstruction du widget et des enfants impliqués.

La structure enfant de l'arborescence n'est pas statique et elle est définie par la description des widgets. Les relations enfants dans les widgets sont ce qui fait l'arborescence de l'interface utilisateur. Pour cela, il existe par composition, il est donc courant de voir les widgets intégrés de Flutter exposant les propriétés *child* ou *children*, en fonction de l'objectif du widget.

L'arborescence des widgets ne fonctionne pas seule dans le framework, elle est aidée de l'arbre des éléments. Cet arbre qui se rapporte à l'arborescence des widgets en représentant le widget construit à l'écran, de sorte que chaque widget aura un élément correspondant dans l'arborescence des éléments après sa construction.

L'arborescence des éléments a une tâche importante dans Flutter. Elle permet de mapper les éléments à l'écran sur l'arborescence des widgets. En outre, elle détermine comment la reconstruction des widgets est effectuée dans les scénarios de mise à jour. Lorsqu'un widget change et doit être reconstruit, cela entraînera une mise à jour de l'élément correspondant. L'élément stocke le type du widget correspondant et une référence à ses éléments enfants. Par exemple, dans le cas du repositionnement d'un widget, l'élément vérifiera le type du nouveau widget correspondant, et en cas de correspondance, il se mettra à jour avec la nouvelle description du widget.

5.2.7 Mon premier programme flutter

Ce n'est pas tout à fait vrai, vous avez déjà mis en œuvre un programme à début et lors de l'installation des composants.

Dans votre environnement de développement Flutter, tel que configuré dans l'annexe, nous pouvons commencer à utiliser les commandes Flutter au travers d'une instruction 'ligne de commande' ou la création d'un nouveau projet dans Visual Studio Code.

5.2.7.1 *Création d'un nouveau projet*

La manière typique de démarrer un projet Flutter est d'exécuter la commande suivante:

```
flutter create <rôle de sortie>
```

Ici, *rôle de sortie* sera également le nom du projet Flutter si vous ne le spécifiez pas comme argument. En exécutant la commande précédente, le dossier avec le nom fourni sera généré avec un exemple de projet Flutter.

Dans un premier temps, il est bon de savoir qu'il existe des options utiles pour manipuler la création de projet flutter. Les principales sont les suivantes :

--org : Ceci peut être utilisé pour changer l'organisation du propriétaire du projet. Si vous connaissez déjà le développement Android ou iOS, il s'agit du nom de domaine inversé. Il est utilisé pour identifier les noms de packages sur Android et comme préfixe dans l'identifiant du bundle iOS. La valeur par défaut est *com.example*.

-s, - sample : la plupart des exemples officiels d'utilisation des widgets ont un identifiant unique que vous pouvez utiliser pour cloner rapidement sur votre machine avec cet argument.

-i, --ios-language et -a, --android-language : ils sont utilisés pour spécifier le type de langage pour le code natif du projet, et ne sont utilisés que si vous prévoyez d'écrire du code de plateforme natif. Nous verrons ultérieurement comment ajouter du code natif au projet.

--project-name : il est utilisé pour changer le nom du projet. Il doit s'agir d'un identifiant de package Dart valide, comme nous l'avons déjà vu lors de l'étude du format pubspec (<https://dart.dev/tools/pub/pubspec>) :

"Les noms de package doivent être tous en minuscules, avec des traits de soulignement pour séparer les mots, `comme_ceci`. Utilisez uniquement des lettres latines de base et des chiffres arabes: [a-z0-9_]. Assurez-vous également que le nom est un identifiant Dart valide - qu'il ne commence pas par des chiffres et n'est pas un mot réservé."

Si vous ne spécifiez pas ce paramètre, il essaie d'utiliser le même nom que le répertoire de sortie. Notez que cet argument doit être le dernier de la liste des arguments fournis.

Je vous propose de créer un répertoire « *AppsFlutter* » sous le répertoire « *tools* ». Lorsque cela est fait, vous allez dans ce répertoire et vous pouvez créer un projet flutter typique par la commande suivante :

```
flutter create bonjour_le_monde
```

```

PS C:\tools\AppsFlutter> flutter create bonjour_le_monde
Creating project bonjour_le_monde...
  bonjour_le_monde\.gitignore (created)
  bonjour_le_monde\idea\libraries\Dart_SDK.xml (created)
  bonjour_le_monde\idea\libraries\kotlinJavaRuntime.xml (created)
  bonjour_le_monde\idea\modules.xml (created)
  bonjour_le_monde\idea\runConfigurations\main_dart.xml (created)
  bonjour_le_monde\idea\workspace.xml (created)
  bonjour_le_monde\.metadata (created)
  bonjour_le_monde\android\app\build.gradle (created)
  bonjour_le_monde\android\app\src\main\kotlin\com\exemple\bonjour_le_monde\MainActivity.kt (created)
  bonjour_le_monde\android\app\build.gradle (created)
  bonjour_le_monde\android\bonjour_le_monde_android.iml (created)
  bonjour_le_monde\android\.gitignore (created)
  bonjour_le_monde\android\app\src\debug\AndroidManifest.xml (created)
  bonjour_le_monde\android\app\src\main\AndroidManifest.xml (created)
  bonjour_le_monde\android\app\src\main\res\drawable\launch_background.xml (created)
  bonjour_le_monde\android\app\src\main\res\mipmap-hdpi\ic_launcher.png (created)
  bonjour_le_monde\android\app\src\main\res\mipmap-mdpi\ic_launcher.png (created)
  bonjour_le_monde\android\app\src\main\res\mipmap-xhdpi\ic_launcher.png (created)
  bonjour_le_monde\android\app\src\main\res\mipmap-xxhdpi\ic_launcher.png (created)
  bonjour_le_monde\android\app\src\main\res\values\styles.xml (created)
  bonjour_le_monde\android\app\src\profile\AndroidManifest.xml (created)
  bonjour_le_monde\android\gradle\wrapper\gradle-wrapper.properties (created)
  bonjour_le_monde\android\gradle.properties (created)
  bonjour_le_monde\android\settings.gradle (created)
  bonjour_le_monde\ios\Runner\AppDelegate.swift (created)
  bonjour_le_monde\ios\Runner\runner-Bridging-Header.h (created)
  bonjour_le_monde\ios\Runner\xcodeproj\project.pbxproj (created)
  bonjour_le_monde\ios\Runner\xcodeproj\xcshareddata\xcschemes\Runner.xcscheme (created)
  bonjour_le_monde\ios\.gitignore (created)
  bonjour_le_monde\ios\Flutter\AppFrameworkInfo.plist (created)
  bonjour_le_monde\ios\Flutter\Debug.xcconfig (created)
  bonjour_le_monde\ios\Flutter\Release.xcconfig (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-1024x1024@1x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-20x20@1x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-20x20@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-20x20@3x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-29x29@1x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-29x29@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-29x29@3x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-40x40@1x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-40x40@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-40x40@3x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-60x60@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-60x60@3x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-76x76@1x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-76x76@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\Icon-App-83.5x83.5@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\LaunchImage.imageset\Contents.json (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@2x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@3x.png (created)
  bonjour_le_monde\ios\Runner\Assets.xcassets\LaunchImage.imageset\README.md (created)
  bonjour_le_monde\ios\Runner\Base.lproj\LaunchScreen.storyboard (created)
  bonjour_le_monde\ios\Runner\Base.lproj>Main.storyboard (created)
  bonjour_le_monde\ios\Runner\Info.plist (created)
  bonjour_le_monde\ios\Runner\xcodeproj\project.xcworkspace\contents.xcworkspacedata (created)
  bonjour_le_monde\ios\Runner\xcodeproj\project.xcworkspace\xcshareddata\IDEWorkspaceChecks.plist (created)
  bonjour_le_monde\ios\Runner\xcodeproj\project.xcworkspace\xcshareddata\workspaceSettings.xcsettings (created)
  bonjour_le_monde\ios\Runner\xcworkspace\contents.xcworkspacedata (created)
  bonjour_le_monde\ios\Runner\xcworkspace\xcshareddata\IDEWorkspaceChecks.plist (created)
  bonjour_le_monde\ios\Runner\xcworkspace\xcshareddata\workspaceSettings.xcsettings (created)
  bonjour_le_monde\lib\main.dart (created)
  bonjour_le_monde\bonjour_le_monde.iml (created)
  bonjour_le_monde\pubspec.yaml (created)
  bonjour_le_monde\README.md (created)
  bonjour_le_monde\test\widget_test.dart (created)
Running "flutter pub get" in bonjour_le_monde...                                         3,3s
Wrote 71 files.

All done!
[✓] Flutter: is fully installed. (Channel stable, 1.20.2, on Microsoft Windows [version 6.1.7601], locale fr-FR)
[✓] Android toolchain - develop for Android devices: is fully installed. (Android SDK version 30.0.2)
[✓] Android Studio: is fully installed. (version 4.0)
[✓] VS Code: 64-bit edition: is fully installed. (version 1.48.0)
[!] Connected device: is not available.

Run "flutter doctor" for information about installing additional components.

In order to run your application, type:

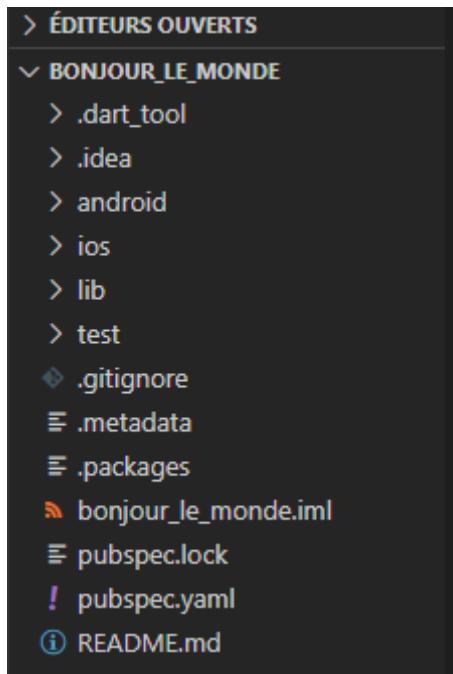
$ cd bonjour_le_monde
$ flutter run

Your application code is in bonjour_le_monde\lib\main.dart.

PS C:\tools\AppsFlutter>

```

Si vous ouvrez ce projet ou ce dossier dans Visual Studio Code, vous obtenez la structure standard d'un projet flutter. Entre nous, elle ressemble beaucoup à la structure d'un projet Dart.



En listant les éléments de structure de base, nous obtenons les éléments suivants :

android/ios : ces répertoires contiennent les codes spécifiques à la plate-forme. Si vous connaissez déjà la structure du projet Android d'Android Studio, il n'y a pas de surprise ici. Il en va de même pour les projets XCode iOS.

bonjour_le_monde.iml : Il s'agit d'un fichier de projet IntelliJ typique, qui contient les informations JAVA_MODULE utilisées par l'EDI.

lib : il s'agit du dossier principal d'une application Flutter; le projet généré doit contenir au moins un fichier main.dart sur lequel commencer l'exécution.

pubspec.yaml et **pubspec.lock** : Si vous vous souvenez d'un ancien paragraphe, ce fichier *pubspec.yaml* est ce qui définit un package Dart. C'est ce qui se passe ici, et c'est l'un des principaux fichiers du projet. Dans ce fichier, vous listez les dépendances d'application, et dans le cas de Flutter, plus que cela.

README.md : Ce fichier est généralement une description du projet, et il est très courant dans les projets open source.

test : il contient tous les fichiers liés aux tests du projet. Ici, nous pouvons ajouter des tests unitaires, comme nous l'avons vu précédemment, ainsi que des tests de widgets en utilisant des packages spécifiques à Flutter.

5.2.7.2 Le fichier pubspec

Le fichier pubspec de Flutter est similaire à celui vu dans un simple projet Dart. Il contient juste une section supplémentaire pour les configurations spécifiques à Flutter.

Voyons le contenu du fichier `pubspec.yaml` en détail:

```
! pubspec.yaml
1   name: bonjour_le_monde
2   description: A new Flutter project.
3
4   # The following line prevents the package from being accidentally published to
5   # pub.dev using `pub publish`. This is preferred for private packages.
6   publish_to: 'none' # Remove this line if you wish to publish to pub.dev
7
8   # The following defines the version and build number for your application.
9   # A version number is three numbers separated by dots, like 1.2.43
10  # followed by an optional build number separated by a +.
11  # Both the version and the builder number may be overridden in flutter
12  # build by specifying --build-name and --build-number, respectively.
13  # In Android, build-name is used as versionName while build-number used as versionCode.
14  # Read more about Android versioning at https://developer.android.com/studio/publish/versioning
15  # In iOS, build-name is used as CFBundleShortVersionString while build-number used as CFBundleVersion.
16  # Read more about iOS versioning at
17  # https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/
18  version: 1.0.0+1
```

Comme vous le savez déjà, la propriété `name` est définie lorsque nous exécutons la commande « `flutter pub create` », suivie de la description du projet par défaut.

NB : Vous pouvez spécifier la description lors de la commande « `flutter create` » en utilisant l'argument `--description`.

La propriété `version` suit les conventions du package Dart: le numéro de version, plus un numéro de version de build facultatif séparé par `+`. En plus de cela, Flutter vous permet de remplacer ces valeurs pendant la construction.

Ensuite, nous avons la section des dépendances du fichier `pubspec`:

```
environment:
  sdk: ">=2.7.0 <3.0.0"

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: ^0.1.3

dev_dependencies:
  flutter_test:
    sdk: flutter
```

Nous commençons par la propriété d'environnement avec la définition des contraintes de version du SDK Dart. Vous pouvez utiliser la version fournie par l'outil, car elle est également suivie des mises à jour du SDK Flutter. Le SDK Dart est intégré au SDK Flutter, vous n'avez donc pas à les installer séparément.

dependencies : qui commence par la dépendance principale d'une application Flutter, le SDK Flutter lui-même, qui contient de nombreux packages de Flutter. En tant que dépendance supplémentaire, le

générateur ajoute le package *cupertino_icons*, qui contient les icônes utilisées par les widgets Cupertino intégrés à Flutter.

dev_dependencies : contient uniquement la dépendance du package flutter_test fourni par le SDK Flutter lui-même, et contient des extensions spécifiques à Flutter par rapport au package de test Dart déjà connu.

Dans le dernier bloc du fichier, il y a une section dédiée à flutter :

```
# The following section is specific to Flutter.

flutter:

# The following line ensures that the Material Icons font is
# included with your application, so that you can use the icons in
# the material Icons class.
uses-material-design: true

# To add assets to your application, add an assets section, like this:
# assets:
#   - images/a_dot_burr.jpeg
#   - images/a_dot_ham.jpeg

# An image asset can refer to one or more resolution-specific "variants", see
# https://flutter.dev/assets-and-images/#resolution-aware.

# For details regarding adding assets from package dependencies, see
# https://flutter.dev/assets-and-images/#from-packages

# To add custom fonts to your application, add a fonts section here,
# in this "flutter" section. Each entry in this list should have a
# "family" key with the font family name, and a "fonts" key with a
# list giving the asset and other descriptors for the font. For
# example:
# fonts:
#   - family: Schyler
#     fonts:
#       - asset: fonts/Schyler-Regular.ttf
#       - asset: fonts/Schyler-Italic.ttf
#         style: italic
#   - family: Trajan Pro
#     fonts:
#       - asset: fonts/TrajanPro.ttf
#       - asset: fonts/TrajanPro_Bold.ttf
#         weight: 700
#
# For details regarding fonts from package dependencies,
# see https://flutter.dev/custom-fonts/#from-packages
```

Cette section flutter nous permet de configurer les ressources qui sont utilisées par l'application, telles que des images, des polices et un fichier JSON. Généralement, tout fichier de code non source qui aide à la composition de l'application.

uses-material-design : permet intégrer les widgets *Material* fournis par Flutter. Nous pouvons également utiliser des icônes Material Design (<https://material.io/tools/icons/?style=baseline>), qui sont dans un format de police de caractères. Pour que cela fonctionne correctement, nous devons activer cette propriété (définissez-la sur *true*) afin que les icônes soient incluses dans l'application.

assets : cette propriété est utilisée pour lister les chemins de ressources qui seront regroupés avec l'application finale. Consultez le commentaire ci-dessus pour plus de détails sur son utilisation. Les fichiers actifs peuvent être organisés de n'importe quelle manière ; ce qui compte pour Flutter, c'est le chemin d'accès aux fichiers. Vous spécifiez ces chemins par rapport à la racine du projet.

Pour ajouter une image à utiliser plus tard, nous ajoutons le chemin dans la liste, ou si nous voulons ajouter tous les fichiers à l'intérieur du répertoire, nous spécifions simplement le chemin du répertoire. Ceci inclut tous les fichiers à l'intérieur de ce répertoire. Notez le caractère '/' à la fin.

polices : Cette propriété nous permet d'ajouter des polices personnalisées à l'application.

5.2.7.3 Le fichier lib/main.dart

Le fichier principal du projet est le point d'entrée de l'application Flutter:

```
void main() {  
  runApp(MyApp());  
}
```

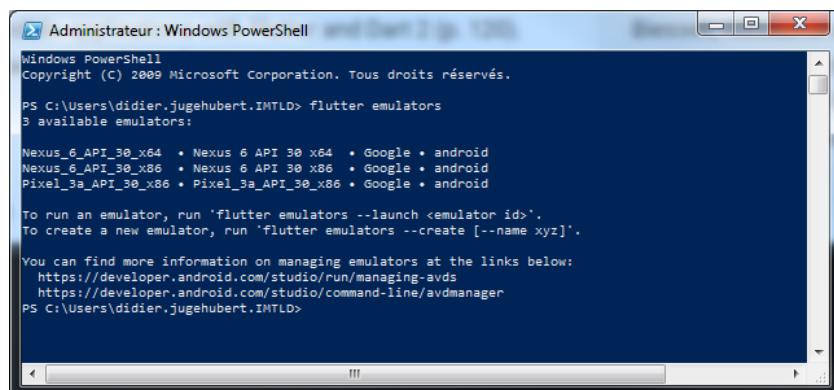
La fonction principale en elle-même est le point d'entrée d'une application Dart. Ce qui fait que l'application Flutter exécutera la fonction *runApp* en passant le widget racine en paramètre (l'application elle-même).

5.2.7.4 Exécution du projet généré

Le projet généré utilise le modèle Flutter par défaut pour créer le projet. Cette application test dispose d'un compteur pour démontrer le style de programmation *React* dans Flutter. Dans l'exemple « *bonjour_le_monde* » que nous avons créé précédemment à l'aide de la commande « *flutter create* », *MyApp* est le widget racine de l'application.

Pour exécuter l'application Flutter, nous devons avoir un appareil ou un simulateur connecté. La vérification est effectuée en utilisant les outils déjà connus de « *flutter doctor* » et de « *flutter emulators* ».

La commande suivante vous permet de connaître les émulateurs Android et iOS existants qui peuvent être utilisés pour exécuter le projet. Vous obtiendrez quelque chose de similaire à la capture d'écran suivante :



```
Administrator : Windows PowerShell  
Windows PowerShell  
Copyright (C) 2009 Microsoft Corporation. Tous droits réservés.  
PS C:\Users\didier.jugehubert.IMTLD> flutter emulators  
3 available emulators:  
Nexus_6_API_30_x64 * Nexus 6 API 30 x64 * Google * android  
Nexus_6_API_30_x86 * Nexus 6 API 30 x86 * Google * android  
Pixel_3a_API_30_x86 * Pixel_3a_API_30_x86 * Google * android  
  
To run an emulator, run 'flutter emulators --launch <emulator id>'.  
To create a new emulator, run 'flutter emulators --create [--name xyz]'.  
  
You can find more information on managing emulators at the links below:  
  https://developer.android.com/studio/run/managing-avds  
  https://developer.android.com/studio/command-line/avdmanager  
PS C:\Users\didier.jugehubert.IMTLD>
```

Comme vous pouvez le lire dans la fenêtre précédente, pour exécuter un émulateur, vous devez taper :

```
flutter emulators --launch <emulateur id>
```

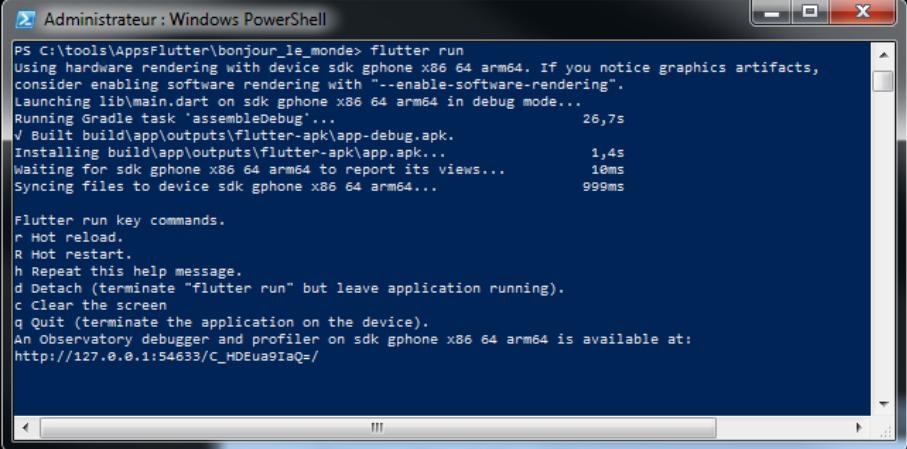
Soit dans notre cas pour une machine 64 bits :

```
flutter emulators --launch Nexus_6_API_30_x64
```

Après avoir lancé l'émulateur, vous pouvez ensuite exécuter l'application par la commande en vous plaçant dans le répertoire racine de votre projet.

```
cd C:\tools\AppsFlutter\bonjour_le_monde  
flutter run
```

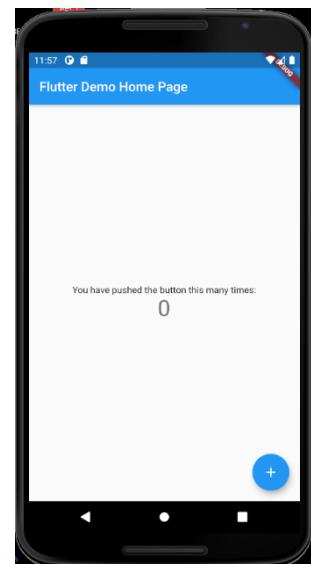
Vous obtenez sur la fenêtre de commande :



```
PS C:\tools\AppsFlutter\bonjour_le_monde> flutter run  
Using hardware rendering with device sdk gphone x86 64 arm64. If you notice graphics artifacts,  
consider enabling software rendering with "-enable-software-rendering".  
Launching lib\main.dart on sdk gphone x86 64 arm64 in debug mode...  
Running Gradle task 'assembleDebug'...  
V Built build\app\outputs\flutter-apk\app-debug.apk.  
Installing build\app\outputs\flutter-apk\app.apk...  
Waiting for sdk gphone x86 64 arm64 to report its views...  
Syncing files to device sdk gphone x86 64 arm64...  
  
Flutter run key commands.  
r Hot reload.  
R Hot restart.  
h Repeat this help message.  
d Detach (terminate "flutter run" but leave application running).  
c Clear the screen  
q Quit (terminate the application on the device).  
An Observatory debugger and profiler on sdk gphone x86 64 arm64 is available at:  
http://127.0.0.1:54633/C_HDEua9IaQ=/
```

Cette commande démarre le débogueur et rend la fonctionnalité de recharge à chaud disponible, comme vous pouvez le voir. La première exécution de l'application peut prendre un peu plus de temps que les exécutions suivantes.

L'application est opérationnelle et vous pouvez voir une marque de débogage dans le coin supérieur droit. Cela signifie que ce n'est pas une version en cours d'exécution, comme vous le savez déjà. Il s'agit de la version de développement de l'application, avec des fonctions de recharge à chaud et de débogage.



6 Interface utilisateur de flutter - Tout est widget

Vous allez découvrir la manière de travailler avec l'interface utilisateur de flutter, la saisie des données utilisateur et les ressources disponibles pour créer des interfaces utilisateur riches.

Vous aborderez les concepts suivants

- ✚ Widgets: Création d'écrans dans flutter
- ✚ Gestion des entrées et des gestes de l'utilisateur
- ✚ Thème et style
- ✚ Navigation entre les écrans

6.1 Widgets : Création d'écrans dans flutter

Vous allez découvrir les concepts centraux des widgets, les différences entre les widgets sans état *stateless* et avec état *stateful*, les widgets les plus courants dans flutter et comment les ajouter à votre application, et comment créer des interfaces complètes à partir de widgets intégrés ou personnalisés développés par vous-même.

6.1.1 Les widgets sans état *stateless* et avec état *stateful*

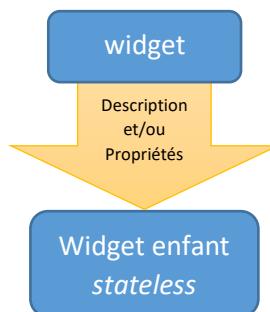
Vous avez vu que les widgets jouent un rôle important dans le développement d'applications flutter. Ce sont les éléments qui construisent l'interface utilisateur. Ils sont la représentation codée de ce qui est visible pour l'utilisateur.

Les interfaces utilisateur ne sont presque jamais statiques, ils changent fréquemment, comme vous le savez. Bien qu'immuables par définition, les widgets ne sont pas censés être définitifs - après tout, nous avons affaire à une interface utilisateur, et une interface utilisateur changera certainement au cours du cycle de vie de toute application. C'est pourquoi flutter nous fournit deux types de widgets: *stateful* et *stateless*.

La grande différence entre ces derniers réside dans la manière dont le widget est construit. Il est de la responsabilité du développeur de choisir le type de widget à utiliser dans chaque situation lors de la composition de l'interface utilisateur, afin de tirer le meilleur parti de la puissance de la couche de rendu de widget de flutter.

6.1.1.1 Les widgets *stateless*

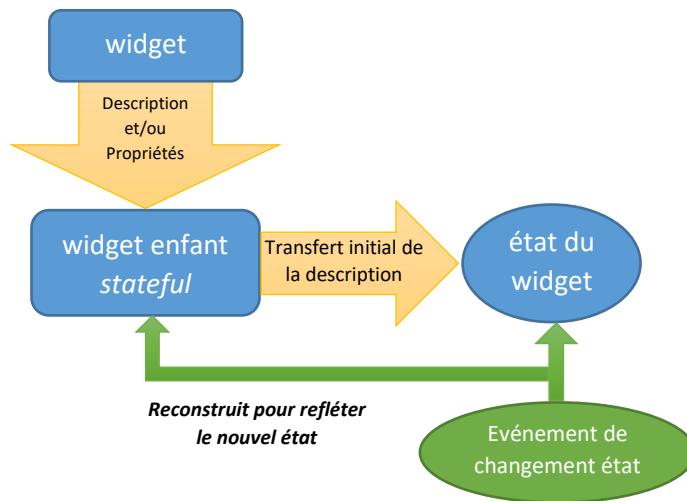
Une interface utilisateur typique sera composée de nombreux widgets, et certains d'entre eux ne changeront jamais leurs propriétés après avoir été instanciés. Ils n'ont pas d'état. C'est-à-dire qu'ils ne changent pas d'eux-mêmes par une action ou un comportement interne. Au lieu de cela, ils sont modifiés par des événements externes sur les widgets parents dans l'arborescence des widgets. Donc, nous pouvons dire que les widgets *stateless* donnent le contrôle de la façon dont ils sont construits sur un widget parent dans l'arborescence. Ce qui suit est une représentation d'un widget *stateless* :



Ainsi, le widget enfant recevra sa description du widget parent et ne le changera pas de lui-même. En termes de code, cela signifie que les widgets sans état n'ont des propriétés définies que lors de la construction, et c'est la seule chose qui doit être créée sur l'écran de l'appareil.

6.1.1.2 Les widgets *stateful*

Contrairement aux widgets *stateless*, qui reçoivent une description de leurs parents qui est maintenue pendant toute sa durée de vie, les widgets *stateful* sont censés changer leurs descriptions de manière dynamique au cours de leur vie. Par définition, les widgets *stateful* sont également immuables, mais ils ont une classe *State* qui représente l'état actuel du widget. Elle est illustrée dans le schéma suivant :



En conservant l'état du widget dans un objet *State* séparé, le framework peut le reconstruire chaque fois que nécessaire sans perdre son état associé actuel. L'élément dans l'arborescence des éléments contient une référence du widget correspondant ainsi que l'objet *State* qui lui est associé. L'objet *State* notifiera lorsque le widget doit être reconstruit et provoquera également une mise à jour dans l'arborescence des éléments.

6.1.1.3 Utilisation des widgets *stateless*

Si vous examinez les widgets *stateless* dans le code. Le tout premier widget *stateless* de l'application est la classe d'application elle-même:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ), // ThemeData
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    ); // MaterialApp
}
```

Comme vous pouvez le voir, la classe *MyApp* étend la classe *StatelessWidget* et remplace la méthode *build(BuildContext)*. Cette méthode décrit une partie de l'interface utilisateur. Autrement dit, il

construit le sous-arbre des widgets en dessous. Dans le cas précédent, *MyApp* est la racine de l'arborescence des widgets et, par conséquent, il construit tous les widgets dans l'arborescence. Dans ce cas, son enfant direct est *MaterialApp*. Selon la documentation, cela est défini comme suit : *BuildContext* est un argument fourni à la méthode de construction comme moyen utile d'interagir avec l'arborescence de widgets qui permet d'accéder à des informations ancestrales importantes qui aident à décrire le widget en cours de construction. N'oubliez pas que la description dépend uniquement de ces informations contextuelles et des propriétés du widget définies dans le constructeur.

En plus d'autres propriétés, *MaterialApp* contient la propriété *home*, qui spécifie le premier widget affiché comme page d'accueil de l'application. Ici, *home* est le widget *MyHomePage*, qui est le widget *stateful* de cet exemple.

6.1.1.4 Utilisation des widgets *stateful*

Le widget *MyHomePage* dans le code est un widget *stateful*, et il est donc défini avec un objet *State* : *_MyHomePageState*, qui contient des propriétés qui affectent l'apparence du widget :

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

En étendant *statefulWidget*, *MyHomePage* doit retourner un objet *State* valide dans sa méthode *createState()*. Dans notre exemple, il renvoie une instance de *_MyHomePageState*.

De plus, l'état est généralement privé pour la bibliothèque de widgets, car les clients externes n'ont pas besoin d'interagir directement avec lui. La classe *_MyHomePageState* suivante représente l'objet *State* du widget *MyHomePage*:

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

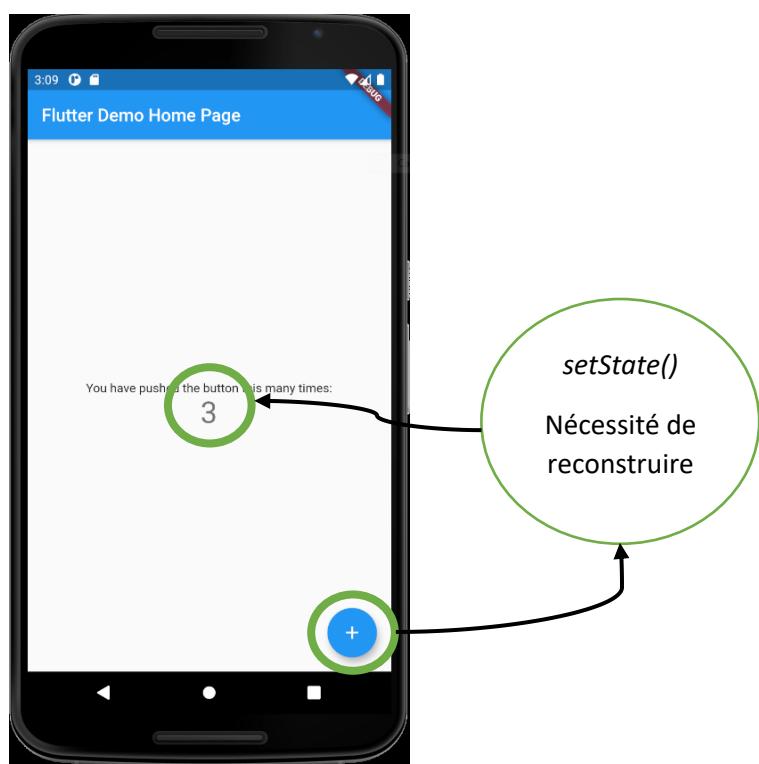
  <> void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    <> return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '_$counter',
              style: Theme.of(context).textTheme.headline4,
            ), // Text
          ], // <Widget>[]
      ), // Column
    ), // Center
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: Icon(Icons.add),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  ); // Scaffold
}
```

Un état de widget valide est une classe qui étend la classe *State* du framework, qui est définie dans la documentation comme suit : "La logique et l'état interne d'un *statefulWidget*."

L'état du widget *MyHomePage* est défini par une seule propriété, *_counter*. La propriété *_counter* conserve le nombre d'appui sur le bouton d'incrémentation dans le coin inférieur droit de l'écran. Cette fois, la classe descendante du widget *State* est responsable de la construction du widget. Il est composé d'un widget *Text* qui affiche la valeur *_counter*. Le texte est un widget intégré utilisé pour afficher du texte à l'écran.

Un widget avec état est censé changer son apparence au cours de sa vie - c'est-à-dire que ce qui le définit changera - et il doit donc être reconstruit pour refléter ces changements. Ici, le changement se produit dans la méthode *_incrementCounter()*, qui est appelée chaque fois que le bouton d'incrémentation est appuyé.

Notez l'utilisation de la propriété *onPressed* du widget *FloatingActionButton*. *FloatingActionButton* est le bouton d'action flottant de *Material Design*, et cette propriété reçoit un appel de la fonction qui sera exécuté lors de l'appui sur le bouton :

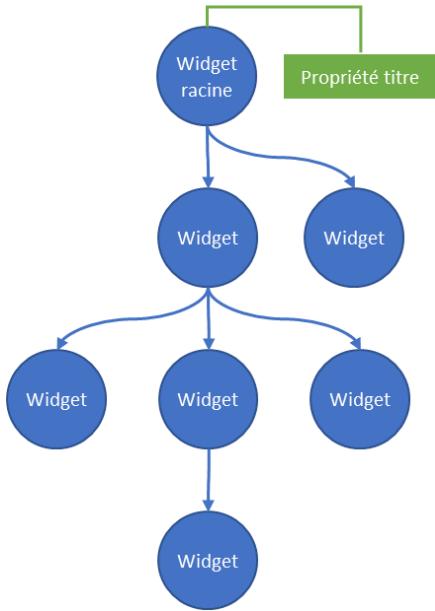


Comment le framework sait-il quand quelque chose dans le widget change et qu'il doit le reconstruire ? *setState* est la réponse. Cette méthode reçoit une fonction en tant que paramètre dans lequel vous devez mettre à jour l'état correspondant du widget (c'est-à-dire la méthode *_incrementCounter*). En appelant *setState*, le framework est notifié qu'il doit reconstruire le widget. Dans l'exemple précédent, il est appelé pour refléter la nouvelle valeur de la propriété *_counter*.

6.1.1.5 Widgets hérités

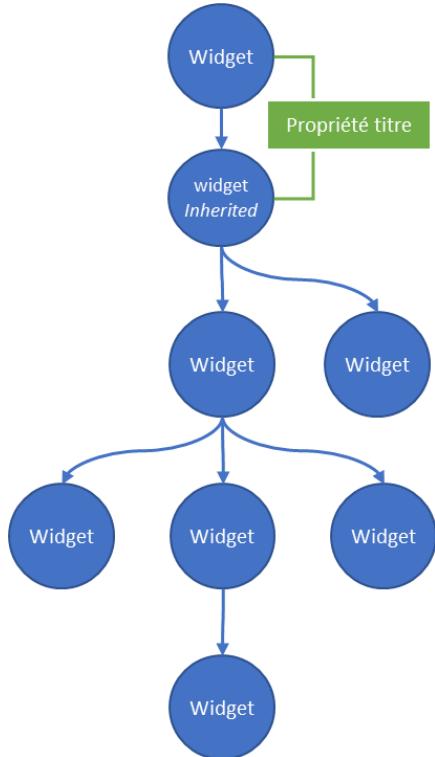
Outre les widgets *stateless* et *stateful*, il existe un autre type de widget dans le framework flutter, les widgets *Inherited*. Parfois, un widget peut avoir besoin d'avoir accès aux données dans l'arborescence, et dans un tel cas, nous aurions besoin de répliquer les informations jusqu'au widget intéressé.

Ce processus est illustré dans le diagramme suivant :



Supposons que certains des widgets situés dans l'arborescence aient besoin d'accéder à la propriété *titre* depuis le widget racine. Pour ce faire, avec les widgets *stateless* ou *stateful*, nous aurions besoin de répliquer la propriété dans les widgets correspondants et de la transmettre via le constructeur. Il peut être ennuyeux de répliquer la propriété sur tous les widgets enfants afin que la valeur atteigne le widget intéressé.

Pour résoudre ce problème, flutter fournit la classe *InheritedWidget*, un type de widget auxiliaire qui aide à propager les informations dans l'arborescence, comme illustré dans le diagramme suivant :



En ajoutant un widget *InheritedWidget* à l'arborescence, tout widget en dessous peut accéder aux données qu'il expose en utilisant la méthode *inheritFromWidgetOfExactType<InheritedWidget>* de la classe *BuildContext* qui reçoit un type *InheritedWidget* comme paramètre et utilise l'arborescence pour trouver le premier widget parent du type demandé.

6.1.1.6 Propriété key d'un widget

Si vous regardez les deux constructeurs des classes *statelessWidget* et *statefulWidget*, vous remarquerez un paramètre nommé *key*. C'est une propriété importante pour les widgets dans Flutter. Cette propriété aide au rendu de l'arborescence des widgets vers l'arborescence des éléments. Outre le type et une référence au widget correspondant, cet élément contient également la clé qui identifie le widget dans l'arborescence. La propriété *key* aide à préserver l'état d'un widget entre les reconstructions. L'utilisation la plus courante de la clé est lorsque nous traitons des collections de widgets qui ont le même *type*. Ainsi, sans clés, l'arborescence des éléments ne saurait pas quel état correspond à quel widget, car ils auraient tous le même type. Par exemple, chaque fois qu'un widget change de position ou de niveau dans l'arborescence des widgets, la mise en correspondance est effectuée dans l'arborescence des éléments pour voir ce qui doit être mis à jour à l'écran pour refléter la nouvelle structure de widget. Lorsqu'un widget a un état, il a besoin de l'état correspondant pour être déplacé avec lui. En bref, c'est ce qu'une clé aide à faire. En maintenant la valeur de clé, l'élément en question connaîtra l'état du widget correspondant qui doit être avec lui.

6.1.2 Les widgets intégrés

Flutter se concentre beaucoup sur l'interface utilisateur, et pour cette raison, il contient un large catalogue de widgets pour permettre la construction d'interfaces personnalisées en fonction de vos besoins. Les widgets disponibles dans flutter vont des widgets simples, tels que le widget *Text*, aux widgets complexes qui aident à concevoir une interface utilisateur dynamique avec des animations et une gestion de plusieurs gestes.

6.1.2.1 Les widgets de base

Les widgets de base de flutter sont un bon point de départ, non seulement pour leur facilité d'utilisation, mais aussi parce qu'ils démontrent la puissance et la flexibilité du framework, même dans des cas simples. Nous ne verrons pas tous les widgets disponibles car cela serait beaucoup trop long. Nous n'en énumérerons donc que certains pour votre connaissance et nous en utiliserons certains dans la pratique afin que vous puissiez apprendre les bases pour en explorer davantage.

6.1.2.2 Le widget *Text*

Il affiche une chaîne de caractère permettant d'appliquer un style :

```
Text("Ceci est un texte");
```

Les principales propriétés du widget *Text* sont les suivantes :

style : classe qui contient le style du texte. Elle expose des propriétés qui permettent de changer la couleur du texte, l'arrière-plan, la famille de polices (permettant l'utilisation d'une police personnalisée à partir des ressources), la hauteur de la ligne, la taille de la police, etc.

textAlign : contrôle l'alignement horizontal du texte, en donnant des options telles que centré ou justifié, par exemple.

maxLines : permet de spécifier un nombre maximum de lignes pour le texte qui sera tronqué si la limite est dépassée.

overflow : définira comment le texte sera tronqué en cas de débordement, en donnant des options telles que la spécification d'un nombre max de lignes. Cela peut être en ajoutant une « ... » à la fin, par exemple.

6.1.2.3 Le widget *Image*

Image affiche une image provenant de différentes sources en différents formats. À partir de la documentation, les formats d'image pris en charge sont JPEG, PNG, GIF, GIF animé, WebP, WebP animé, BMP et WBMP:

```
Image(  
    image:AssetImage ("assets/dart_logo.jpg"),  
) ;
```

La propriété *image* du widget spécifie *ImageProvider*. L'image à afficher peut provenir de différentes sources. De ce fait, la classe *Image* contient différents constructeurs pour différentes manières de charger des images :

Image.image pour obtenir une image à partir *ImageProvider*.

Image.asset crée *AssetImage*, qui permet d'obtenir une image à partir des ressources à l'aide d'une clé. Par exemple :

```
Image.asset("assets/dart_logo.jpg");
```

Image.network crée *NetworkImage*, qui permet d'obtenir une image à partir d'une URL. Par exemple :

```
Image.network("https://picsum.photos/250?image=9");
```

Image.file crée *FileImage*, qui permet d'obtenir une image à partir d'un fichier. Par exemple :

```
Image.file(file_path);
```

Image.memory crée *MemoryImage*, qui permet d'obtenir une image d'un tableau mémoire de type *Uint8List*. Par exemple :

```
Image.memory(Uint8List(images_bytes));
```

La classe *Image* a d'autres propriétés communes qui sont :

height/width : permet de spécifier la taille de l'image.

repeat : permet de répéter l'image pour couvrir l'espace disponible.

alignement : Pour aligner l'image dans une position spécifique dans ses limites de l'affichage.

fit : Pour spécifier comment l'image doit être inscrite dans l'espace disponible.

6.1.2.4 Les widgets Material Design et iOS Cupertino

De nombreux widgets de flutter descendent en quelque sorte d'une directive spécifique à la plate-forme : *Material Design* ou *iOS Cupertino*. Cela aide le développeur à suivre les directives spécifiques à la plate-forme de la manière la plus simple possible.

Flutter, par exemple, n'a pas de widget *Button*; au lieu de cela, il fournit des implémentations alternatives de boutons pour Google Material Design et iOS Cupertino.

Nous n'allons pas approfondir chaque propriété ou comportement de widget, car ceux-ci peuvent être facilement étudiés en exécutant des exemples ou en visitant la documentation. Vous pouvez également consulter l'application Flutter Gallery sur Google Play (<https://play.google.com/store/apps/details?id=io.flutter.demo.gallery>) pour trouver une courte et intéressante démonstration des widgets disponibles.

6.1.2.5 Les boutons

Comme indiqué dans le paragraphe, flutter n'implémente pas de widget bouton mais implémente un bouton suivant le type de plateforme.

Du côté *Material Design*, flutter implémente les composants de bouton suivants:

RaisedButton : Un bouton en relief se compose d'une forme rectangulaire qui plane sur l'arrière-plan.

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

FlatButton : un bouton plat est une section imprimée sur un widget qui réagit aux touches en se modifiant au travers de la couleur.

IconButton : un bouton type icône est une image imprimée sur un widget qui réagit aux touches.

Autre classe de *Material Design*, *Ink*, sur le site Web des directives de conception de widget, peut être expliquée comme suit:

"Composant qui fournit une action radiale sous la forme d'une ondulation visuelle s'étendant vers l'extérieur à partir du toucher de l'utilisateur."

DropDownButton : Affiche l'élément actuellement sélectionné et une flèche qui ouvre un menu pour sélectionner un autre élément.

PopUpMenuItem : Affiche un menu lorsque vous appuyez dessus.

Pour le style *iOS Cupertino*, flutter fournit la classe *CupertinoButton*.

6.1.2.6 Le canevas (Scaffold)

Le canevas implémente la structure de base d'une mise en page dans *Material Design* ou *iOS Cupertino*.

Pour *Material Design*, le widget *Scaffold* peut contenir plusieurs composants de *Material Design* :

body : Le corps est le contenu principal du canevas et il est affiché sous *AppBar*.

AppBar : une barre d'applications se compose d'une barre d'outils et potentiellement d'autres widgets.

TabBar : un widget *Material Design* qui affiche une ligne horizontale d'onglets. Celui-ci est généralement utilisé dans le cadre de la barre d'applications.

TabBarView : Une vue qui affiche le widget qui correspond à l'onglet actuellement sélectionné. Généralement utilisé en conjonction avec *TabBar* et utilisé comme widget de corps principal.

BottomNavigationBar : Les barres de navigation inférieures facilitent l'exploration et le basculement entre les vues de niveau supérieur en un seul clic.

Drawer : Un panneau *Material Design* qui se glisse horizontalement à partir du bord d'un canevas pour afficher les liens de navigation dans une application.

Dans *iOS Cupertino*, la structure est différente avec des transitions et des comportements spécifiques. Les classes iOS Cupertino disponibles sont *CupertinoPageScaffold* et *CupertinoTabScaffold*, qui sont généralement composées des éléments suivants:

CupertinoNavigationBar : une barre de navigation supérieure. Elle est généralement utilisée avec *CupertinoPageScaffold*.

CupertinoTabBar : Une barre d'onglets inférieure généralement utilisée avec *CupertinoTabScaffold*.

6.1.2.7 Les dialogues

Les dialogues *Material Design* et *iOS Cupertino* sont implémentés par flutter. Du côté de *Material Design*, il s'agit de *SimpleDialog* et *AlertDialog*; du côté de *iOS Cupertino*, ce sont *CupertinoDialog* et *CupertinoAlertDialog*.

6.1.2.8 Les champs de texte

Les champs de texte sont également implémentés dans les deux directives, par le widget *TextField* dans *Material Design* et par le widget *CupertinoTextField* dans *iOS Cupertino*. Les deux affichent le clavier pour l'entrée utilisateur.

Certaines de leurs propriétés communes sont les suivantes:

autofocus : si le *TextField* doit être sélectionné automatiquement (si rien d'autre n'est déjà sélectionné)

enabled : pour définir le champ comme modifiable ou non.

keyboardType : pour changer le type de clavier affiché à l'utilisateur lors de l'édition.

6.1.2.9 Les widgets de sélection

Les widgets de contrôle disponibles pour la sélection dans *Material Design* sont les suivants :

Checkbox : La case à cocher permet la sélection de plusieurs options dans une liste.

Radio : Le bouton radio permet une seule sélection dans une liste d'options.

Switch : L'interrupteur (*switch*) permet de basculer (marche / arrêt) une seule option.

Slider : Le curseur permet la sélection d'une valeur dans une plage en déplaçant le curseur.

Du côté *iOS Cupertino*, certaines de ces fonctionnalités de widget n'existent pas. Cependant, il existe quelques alternatives disponibles:

CupertinoActionSheet : Une feuille d'action modale de style iOS pour choisir une option parmi d'autres.

CupertinoPicker : également un contrôle de sélection. Il est utilisé pour sélectionner un élément dans une courte liste.

CupertinoSegmentedControl : se comporte comme un bouton radio, où la sélection est un élément unique d'une liste d'options.

CupertinoSlider : similaire à Slider.

CupertinoSwitch : Ceci est également similaire au Switch de *Material Design*.

6.1.2.10 Les sélecteurs de date et d'heure

Pour *Material Design*, flutter fournit des sélecteurs de date et d'heure via les fonctions *showDatePicker* et *showTimePicker*, qui créent et affiche la boîte de dialogue *Material Design* pour les actions correspondantes.

Du côté *iOS Cupertino*, les widgets *CupertinoDatePicker* et *CupertinoTimerPicker* sont fournis, suivant le style *CupertinoPicker* précédent.

6.1.3 L'utilisation des widgets intégrés

Certains widgets semblent ne pas apparaître à l'écran pour l'utilisateur, mais s'ils sont dans l'arborescence des widgets, ils y seront d'une manière ou d'une autre, affectant l'apparence d'un widget enfant (comme la façon dont il est positionné ou stylisé). Pour positionner un bouton dans le coin inférieur de l'écran, par exemple, nous pourrions spécifier une position liée à l'écran, mais comme vous l'avez peut-être remarqué, les boutons et autres widgets n'ont pas de propriété *Position*.

Alors, vous vous demandez peut-être : "Comment les widgets sont-ils organisés à l'écran ?" La réponse est encore une fois les widgets. C'est vrai ! flutter fournit des widgets pour composer la mise en page elle-même, avec positionnement, dimensionnement, style, etc.

6.1.3.1 Les conteneurs

L'affichage d'un seul widget à l'écran n'est pas un bon moyen d'organiser une interface utilisateur. Nous dresserons généralement une liste de widgets organisés d'une manière spécifique. Pour ce faire, nous utilisons des widgets conteneurs.

Les conteneurs les plus courants dans Flutter sont les widgets *Row* et *Column*. Ils ont une propriété *children* qui s'attend à ce qu'une liste de widgets soit affichée dans une direction spécifique (c'est-à-dire une liste horizontale pour *Row* ou une liste verticale pour *Column*).

Un autre widget largement utilisé est le widget *Stack*, qui organise les enfants en couches, où un enfant peut chevaucher un autre enfant partiellement ou totalement. Si vous avez déjà développé une application mobile, vous avez peut-être déjà utilisé des listes et des grilles. Flutter fournit des classes pour les deux : à savoir, les widgets *ListView* et *GridView*. En outre, d'autres widgets de conteneur moins typiques mais néanmoins importants sont disponibles, tels que *Table*, par exemple, qui organise les enfants dans une mise en page tabulaire.

6.1.3.2 La style et le positionnement

La tâche de positionnement d'un widget enfant dans un conteneur, tel qu'un widget *Stack*, par exemple, est effectuée à l'aide d'autres widgets. Flutter fournit des widgets pour des tâches très spécifiques. Le centrage d'un widget à l'intérieur d'un conteneur se fait en l'enveloppant dans un widget *Center*. L'alignement d'un widget enfant par rapport à un parent peut être effectué avec le widget *Align*, où vous spécifiez la position souhaitée via sa propriété d'alignement. Un autre widget utile est *Padding*, qui nous permet de spécifier un espace autour de l'enfant donné. Les fonctionnalités de ces widgets sont regroupées dans le widget *Container*, qui combine ces widgets communs de positionnement et de style pour les appliquer directement à un enfant, ce qui rend le code beaucoup plus propre et plus court.

6.1.3.3 Les autres widgets (gestes, animations et transformations)

Flutter fournit des widgets pour tout ce qui concerne l'interface utilisateur. Par exemple, les gestes tels que le défilement ou les touches seront tous liés à un widget qui gère les gestes. Les animations et transformations, telles que la mise à l'échelle et la rotation, sont également toutes gérées par des widgets spécifiques.

Nous allons étudier certains d'entre eux en détail dans les chapitres suivants, lorsque nous développerons des parties d'une petite application. Nous ne sommes pas en mesure d'explorer tous les widgets disponibles et toutes les combinaisons possibles d'entre eux.

6.1.4 La création de widgets personnalisés

Lors de la création d'interfaces utilisateur avec flutter, nous devrons toujours créer des widgets personnalisés. Nous ne pouvons et ne voulons pas y échapper. Après tout, la composition de widgets pour créer des interfaces uniques est ce que flutter permet le mieux.

6.1.5 Ma deuxième application flutter

Pour mieux apprêhender les concepts d'utilisation de flutter, je vous propose de développer une application de gestion de services entre amis. Votre application « service_entre_amis » comprendra deux écrans. Dans les deux cas, vous utiliserez les composants *Material Design* fournis par Flutter. Le premier écran sera une liste des services en cours, et le second sera un formulaire pour demander un service à un ami. Pour l'instant, vous utiliserez des listes en mémoire. Autrement dit, les informations ne seront stockées nulle part ailleurs que dans l'application.

6.1.5.1 La mise en page de l'application

Dans un premier temps, le code de l'application ne sera pas entièrement fonctionnel mais il sera suffisamment petit pour afficher la mise en page. Il crée une instance de widget *MaterialApp* qui définit l'écran d'accueil sur la page de liste des services, appelée *PageServices*:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Service entre amis',
      home: PageServices(
        servicesDemandes: simulServicesDemandes,
        servicesTermimes: simulServicesTermimes,
        servicesRefuses: simulServicesRefuses,
        servicesAcceptes: simulServicesAcceptes),
    );
  }
}
```

MaterialApp est un widget qui fournit des outils utiles pour l'ensemble de l'application. L'un d'eux est le widget *Theme*, qui vous permettra de modifier les styles et les couleurs de nos applications en suivant les directives de *Material Design*. Un autre outil utile est le widget *Navigator*, qui gère un ensemble de widgets d'application à la manière d'une pile de navigation, où vous pourrez naviguer vers un écran en le poussant (*push*) sur le navigateur ou en revenant en arrière (*pop*). Nous utiliserons les deux widgets dans l'application.

Vous avez déjà implémenté *Navigator* lorsque vous définissez la propriété *home* du widget *MaterialApp*. *Navigator* permet de définir des routes spécifiques pointant vers des widgets, et lorsque nous naviguons vers une route, il sera capable de naviguer vers le widget correspondant. En définissant la propriété *home* dans un widget, vous définissez la racine du widget *Navigator*.

MaterialApp est un widget qui fournit des outils utiles pour l'ensemble de l'application. L'un d'eux est le widget *Theme*, qui vous permettra de modifier les styles et les couleurs de nos applications en suivant les directives de *Material Design*. Un autre outil utile est le widget *Navigator*, qui gère un ensemble de widgets d'application à la manière d'une pile de navigation, où vous pourrez naviguer vers un écran en le poussant (*push*) sur le navigateur ou en revenant en arrière (*pop*). Nous utiliserons les deux widgets dans l'application.

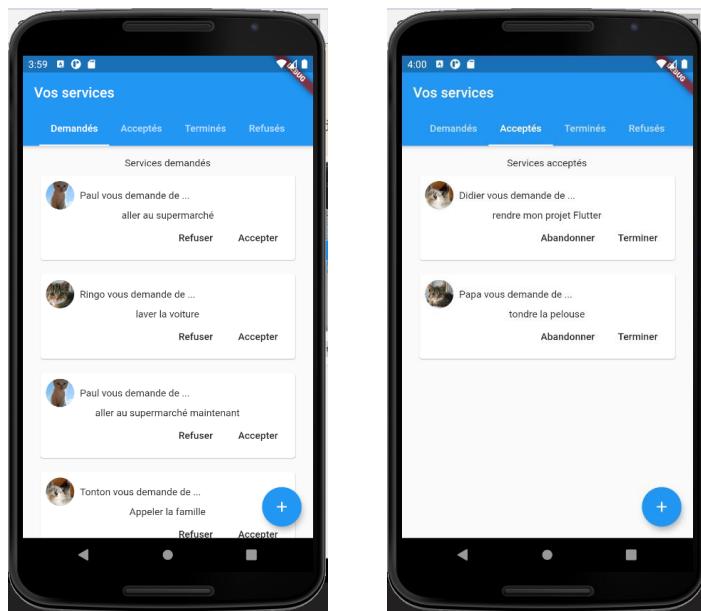
Vous avez déjà implémenté *Navigator* lorsque vous définissez la propriété *home* du widget *MaterialApp*. *Navigator* permet de définir des routes spécifiques pointant vers des widgets, et lorsque nous naviguons vers une route, il sera capable de naviguer vers le widget correspondant. En définissant la propriété *home* dans un widget, vous définissez la racine du widget *Navigator*. Comme vous pouvez le voir, le widget *PageServices* a des paramètres définis pour le constructeur.

6.1.5.2 La page d'accueil de l'application

6.1.5.2.1 La mise en page

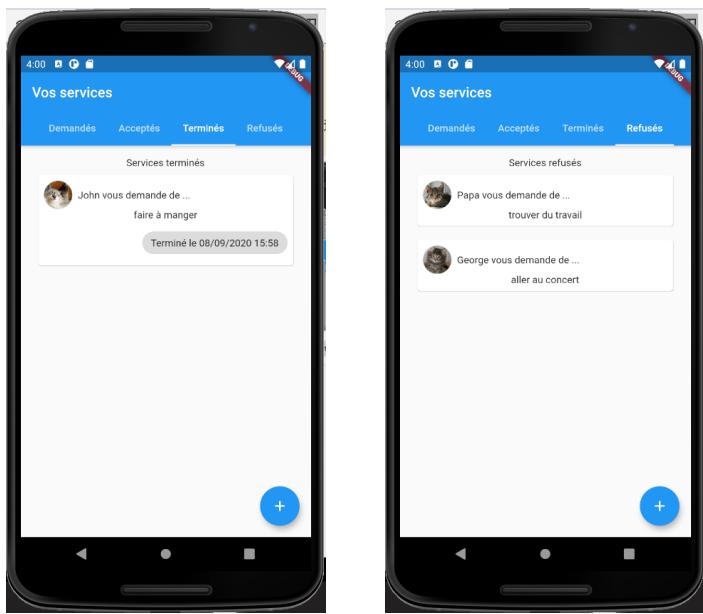
La page d'accueil de l'application est un écran qui sera composé de quatre onglets listant les services et leurs statuts :

- ⊕ **Services demandés** : Ce sont les services demandés par des amis auxquels nous n'avons pas encore répondu.
- ⊕ **Services acceptés** : Ce sont les services que nous avons acceptés et qui sont en cours en ce moment.



Services terminés : Les services déjà complétés

Services refusés : La liste des services que nous avons refusé de faire.



La liste contiendra tous les services de l'application, séparées par catégories. En haut de la mise en page, nous avons une instance *TabBar* qui sera utilisée pour changer l'onglet dans la liste souhaitée. Suite à cela, sur chaque onglet, nous avons une liste d'éléments, qui contiennent les actions correspondant à la catégorie choisie.

Nous avons créé des classes *Amis* et *Service* pour représenter les données de l'application. Ici, il s'agit de classes de données simples qui ne contiennent aucune logique métier avancée.

```
class Amis {
    final String nom;
    final String numero;
    final String photoURL;

    Amis({
        this.nom,
        this.numero,
        this.photoURL,
    });
}
```

```
import 'amis.dart';

class Service {
    final String uuid;
    final String description;
    final DateTime dateEcheance;
    final bool accepte;
    final DateTime complete;
    final Amis amis;

    Service({
        this.uuid,
        this.description,
```

```

    this.dateEcheance,
    this.accepte,
    this.complete,
    this.amis,
  });

get estEnCours => accepte == true && complete == null;
get estDemande => accepte == null;
get estComplete => complete != null;
get estRefuse => accepte == false;
}

```

Enfin, le bouton d'action en bas de l'écran vous permettra d'aller sur la deuxième page « *Demander un service* », où l'utilisateur pourra demander un service à ses amis.

6.1.5.2.2 Le code de la mise en page

Tout d'abord, vous devez définir votre page d'accueil comme une instance *statelessWidget*, car vous voulez gérer la mise en page et vous n'avez, pour le moment, aucune action à gérer qui entraînerait un changement d'état. C'est pourquoi le widget parent, *MyApp*, transmet comme paramètres les quatre listes de services. N'oubliez pas que lorsqu'un widget est sans état, sa description est définie par le widget parent lors de sa création. Ceci est illustré dans le code suivant :

```

class PageServices extends StatelessWidget {
  final List<Service> servicesDemandes;
  final List<Service> servicesTermimes;
  final List<Service> servicesRefuses;
  final List<Service> servicesAcceptes;

  PageServices({
    Key key,
    this.servicesDemandes,
    this.servicesTermimes,
    this.servicesRefuses,
    this.servicesAcceptes,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 4,
      child: Scaffold(
        appBar: AppBar(
          title: Text("Vos services"),
          bottom: TabBar(isScrollable: true, tabs: [
            _construireOnglet("Demandés"),
            _construireOnglet("Acceptés"),
            _construireOnglet("Terminés"),
            _construireOnglet("Refusés"),

```

```

        ],
    ),
    body: TabBarView(
        children: [
            _ListeServices(
                titre: "Services demandés", services: servicesDemandes),
            _ListeServices(
                titre: "Services acceptés", services: servicesAcceptes),
            _ListeServices(
                titre: "Services terminés", services: servicesTermimes),
            _ListeServices(titre: "Services refusés", services: servicesRefuse
s),
        ],
    ),
    floatingActionButton: FloatingActionButton(
        onPressed: () {},
        tooltip: 'Demander un service',
        child: Icon(Icons.add),
    ),
),
);
}
}

Widget _construireOnglet(String titre) {
    return Tab(
        child: Text(titre),
    );
}
}

```

Comme indiqué dans le code précédent, le widget est défini par les listes spécifiques à l'état des services. Notez également le paramètre *key*. Bien que cela ne soit pas vraiment nécessaire ici, il est recommandé de définir le paramètre.

La méthode *build()* est constitué des éléments suivants :

- ⊕ Le premier widget présent dans la sous-arborescence de widget *PageServices* est le widget *DefaultTabController*, qui gère le changement d'onglet pour vous.
- ⊕ Puis un widget *Scaffold*, qui implémente la structure de base de *Material Design*. Ici, vous utiliserez déjà certains de ces éléments, notamment la barre d'application et le bouton d'action. Ce widget *Scaffold* est très utile pour concevoir des applications qui suivent le concept de *Material Design* car il fournit des propriétés utiles basées sur les directives graphiques :
 - *AppBar* : pour ajouter un titre à l'aide d'un widget *Text*. Dans certains cas, vous pouvez également y ajouter des actions ou une mise en page personnalisée. Ici, nous avons ajouté une instance *TabBar* juste en bas de la barre d'application qui affichera les onglets disponibles.
 - *FloatingActionButton* : Pour, vous avez seulement ajouté une icône en utilisant le widget *Icon*, qui contient une icône de *Material Design* fournie par le framework.

- La propriété *body* du widget *Scaffold* est l'endroit où vous allez concevoir la mise en page elle-même. Il est défini comme suit :
 - *TabBarView* : affiche le widget correspondant pour l'onglet sélectionné dans l'instance *DefaultTabController* définie précédemment.
 - La propriété *child* : défini les widgets de la barre d'onglets et renvoie le widget correspondant de chaque onglet.

Les éléments de la barre d'onglets sont créés par la méthode *_construireOnglet()* défini en fin de classe.

Comme vous pouvez le voir, la fonction crée un élément d'onglet par catégorie en créant simplement un sous-arbre *Tab*, où le titre est l'identifiant de l'élément. De la même manière, chaque section de liste de services est définie dans sa propre instance de la classe *_ListeServices()*:

```
class _ListeServices extends StatelessWidget {
  final String titre;
  final List<Service> services;

  const _ListeServices({Key key, this.titre, this.services}):super(key);

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisSize: MainAxisSize.max,
      children: [
        Padding(
          child: Text(titre),
          padding: EdgeInsets.only(top: 16.0),
        ),
        Expanded(
          child: ListView.builder(
            physics: BouncingScrollPhysics(),
            itemCount: services.length,
            itemBuilder: (BuildContext context, int index) {
              final service = services[index];
              return _ElementListeServices(service: service);
            },
          ),
        ],
      );
  }
}
```

Le widget *_ListeServices* est représenté par un widget de colonne qui a deux widgets enfants :

1. Un widget *Text* (avec un parent *Padding*) contenant le titre de la section,
2. Un widget *ListView* qui contiendra chacun des éléments.

Ici, vous avez utilisé le constructeur nommé *ListView.builder()*. Ce constructeur de liste attend des instances *itemCount* et *itemBuilder*, que vous définissez à l'aide de la liste passée en argument dans l'appel à *_ListeServices()* :

- *itemCount* est simplement la taille de la liste.
- *itemBuilder* doit être une fonction qui renvoie le widget correspondant à l'élément dans une position spécifique. Cette fonction reçoit *BuildContext*, comme la méthode *build()* du widget, ainsi qu'une position d'index (ici, vous utilisez l'argument *index* pour obtenir le service correspondant de la liste des services).

Cette forme de création est optimale pour les grandes listes ou des listes qui se développent au cours de l'exécution du programme, ou même les listes à défilement infini (que vous avez peut-être déjà vues dans certaines applications), car elle ne crée des éléments que s'ils sont nécessaires, évitant ainsi le gaspillage de mémoire.

La fonction *itemBuilder* crée un widget *_ElementListeServices* qui est basé sur le widget *Card* pour chaque service dans la liste d'arguments *services* en obtenant l'élément correspondant avec *service = service[index];*.

```
class _ElementListeServices extends StatelessWidget {
  final Service service;

  const _ElementListeServices({Key key, this.service}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      key: ValueKey(service.uuid),
      margin: EdgeInsets.symmetric(vertical: 10.0, horizontal: 25.0),
      child: Padding(
        child: Column(
          children: <Widget>[
            _enteteElement(service),
            Text(service.description),
            _piedDePageElement(service),
          ],
        ),
        padding: EdgeInsets.all(8.0),
      ),
    );
  }
}
```

Lorsque vous utiliserez des éléments de liste, vous aurez toujours besoin d'une clé pour le widget, du moins lorsque vous utiliserez la gestion des événements. Ceci car les listes dans Flutter peuvent recycler de nombreux éléments pendant les événements de défilement, et en ajoutant une clé, vous associez un widget spécifique à un état spécifique.

Une autre propriété du widget *Card* est la propriété *margin*, qui ajoute une marge au widget. Dans ce cas, vous ajoutez une marge de 10,0 en haut et en bas, et une de 25,0 à gauche et à droite. Le corps du widget est divisé en trois parties :

1. *_enteteElement()* : affiche dans l'en-tête l'ami qui a demandé le service.

```

Row _enteteElement(Service service) {
    return Row(
        children: <Widget>[
            CircleAvatar(
                backgroundImage: NetworkImage(
                    service.amis.photoURL,
                ),
            ),
            Expanded(
                child: Padding(
                    padding: EdgeInsets.only(left: 8.0),
                    child: Text('${service.amis.nom} vous demande de ...'),
                ),
            ),
        ],
    );
}

```

L'en-tête est défini comme un sous-arbre avec le widget *Row*[*CircleAvatar*, *Expanded*]. Il comporte une instance *CircleAvatar* qui représente un utilisateur dans une image circulaire. Ici, nous avons utilisé le fournisseur *NetworkImage* avec une image venant de internet en passant simplement une URL. L'espace restant du widget *Row* est utilisé par *Text* avec le nom de l'ami.

2. Le corps de l'élément qui est composé uniquement d'un widget *Text* avec la description du service.
3. *_piedDePageElement* : contient les actions disponibles pour la demande de service en fonction de la catégorie de celui-ci :

```

Widget _piedDePageElement(Service service) {
    if (service.estComplete) {
        final format = new DateFormat("dd/MM/yyyy HH:mm");
        return Container(
            margin: EdgeInsets.only(top: 8.0),
            alignment: Alignment.centerRight,
            child: Chip(
                label: Text('Terminé le ${format.format(service.complete)}'),
            ),
        );
    }
    if (service.estDemande) {
        return Row(
            mainAxisAlignment: MainAxisAlignment.end,
            children: <Widget>[
                FlatButton(
                    child: Text('Refuser'),
                    onPressed: () {},
                ),
                FlatButton(
                    child: Text('Accepter'),
                ),
            ],
        );
    }
}

```

```

        onPressed: () {},
    ),
],
);
}
if (service.estEnCours) {
    return Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: <Widget>[
            FlatButton(
                child: Text('Abandonner'),
                onPressed: () {},
            ),
            FlatButton(
                child: Text('Terminer'),
                onPressed: () {},
            ),
        ],
    );
}
return Container();
}

```

La fonction `_piedDePageElement()` renvoie un widget en fonction du statut du service. Les statuts de service sont définis par les getters de la classe `Service` :

estDemande : Dans ce cas le service n'a pas encore été accepté ou refusé, vous renvoyez un widget `Row` avec deux instances matérielles `FlatButton` avec les actions disponibles correspondantes : **Refuser** ou **Accepter**. `FlatButton` est un bouton *Material Design* qui n'a pas d'élévation ou de couleur d'arrière-plan.

estComplete : Dans ce cas le service a été rendu et est terminé, vous renvoyez un widget `Container` qui affiche la date deréalisation du service dans un widget `Text`.

estEnCours : Dans ce cas le service est en cours de réalisation, vous renvoyez un widget `Row` deux instances matérielles `FlatButton` avec les actions disponibles correspondantes : **Abandonner** ou **Terminer**. Pour l'état *terminé*, vous affichez la date et l'heure d'achèvement du service à l'aide de la classe `DateFormat` de Dart à l'intérieur d'un widget `Chip` pour se différencier du reste du texte.

estRefuse : Dans ce cas par défaut, vous renvoyez un widget `Conteneur` sans contrainte de taille. Il s'agit d'un conteneur vide qui ne prend pas de place sur le *layout*.

Vous pouvez utiliser les méthodes de classe d'assistance `EdgeInsets` chaque fois que vous définissez un remplissage ou une marge.

Pour le moment, vous n'avez gérer aucune action de l'utilisateur.

N.B. : Ci-dessous vous trouverez le code du fichier `service_simules.dart` permettant d'avoir des services définis dans les différents états au démarrage de l'application.

```
import 'amis.dart';
import 'service.dart';
import 'package:uuid/uuid.dart';

final uuid = Uuid();

final simulServicesDemandes = [
  Service(
    uuid: uuid.v4(),
    description: 'aller au supermarché',
    dateEcheance: DateTime.now().add(Duration(days: 1)),
    amis: Amis(
      nom: 'Paul',
      numero: '11111',
      photoURL: "https://placekitten.com/200/300",
    ),
  ),
  Service(
    uuid: uuid.v4(),
    description: 'laver la voiture',
    dateEcheance: DateTime.now().add(Duration(hours: 5)),
    amis: Amis(
      nom: 'Ringo',
      numero: '22222',
      photoURL: "https://placekitten.com/200/286",
    ),
  ),
  Service(
    uuid: uuid.v4(),
    description: 'aller au supermarché maintenant',
    dateEcheance: DateTime.now(),
    amis: Amis(
      nom: 'Paul',
      numero: '11111',
      photoURL: "https://placekitten.com/200/300",
    ),
  ),
  Service(
    uuid: uuid.v4(),
    description: 'Appeler la famille',
    dateEcheance: DateTime.now().add(Duration(hours: 12)),
    amis: Amis(
      nom: 'Tonton',
      numero: '55555',
      photoURL: "https://placekitten.com/200/200",
    ),
  ),
];

```

```

final simulServicesAcceptes = [
  Service(
    uuid: uuid.v4(),
    description: 'rendre mon projet Flutter',
    dateEcheance: DateTime.now().add(Duration(days: 15)),
    accepte: true,
    amis: Amis(
      nom: 'Didier',
      numero: '88888',
      photoURL: "https://placekitten.com/300/300",
    ),
  ),
  Service(
    uuid: uuid.v4(),
    description: 'tondre la pelouse',
    dateEcheance: DateTime.now().add(Duration(hours: 5)),
    accepte: true,
    amis: Amis(
      nom: 'Papa',
      numero: '66666',
      photoURL: "https://placekitten.com/250/250",
    ),
  ),
),
];
final simulServicesTermimes = [
  Service(
    uuid: uuid.v4(),
    description: 'faire à manger',
    dateEcheance: DateTime.now().add(Duration(hours: -2)),
    complete: DateTime.now(),
    accepte: true,
    amis: Amis(
      nom: 'John',
      numero: '33333',
      photoURL: "https://placekitten.com/275/275",
    ),
  ),
],
];
final simulServicesRefuses = [
  Service(
    uuid: uuid.v4(),
    description: 'trouver du travail',
    dateEcheance: DateTime.now().add(Duration(days: 8)),
    accepte: false,
    amis: Amis(
      nom: 'Papa',
      numero: '33333',
      photoURL: "https://placekitten.com/250/250",
    ),
),
];

```

```

),
Service(
  uuid: uuid.v4(),
  description: 'aller au concert',
  dateEcheance: DateTime.now().add(Duration(hours: 8)),
  accepte: false,
  amis: Amis(
    nom: 'George',
    numero: '44444',
    photoURL: "https://placekitten.com/350/350",
  ),
),
],
);

final simulAmis = [
  Amis(
    nom: 'Paul',
    numero: '11111',
    photoURL: "https://placekitten.com/200/300",
  ),
  Amis(
    nom: 'Ringo',
    numero: '22222',
    photoURL: "https://placekitten.com/200/286",
  ),
  Amis(
    nom: 'John',
    numero: '33333',
    photoURL: "https://placekitten.com/275/275",
  ),
  Amis(
    nom: 'George',
    numero: '44444',
    photoURL: "https://placekitten.com/350/350",
  ),
  Amis(
    nom: 'Tonton',
    numero: '55555',
    photoURL: "https://placekitten.com/200/200",
  ),
  Amis(
    nom: 'Papa',
    numero: '66666',
    photoURL: "https://placekitten.com/250/250",
  ),
  Amis(
    nom: 'Maman',
    numero: '77777',
    photoURL: "https://placekitten.com/225/225",
  )
];

```

```
),
Amis(
    nom: 'Didier',
    numero: '88888',
    photoURL: "https://placekitten.com/300/300",
),
];

```

N.B. : Le package « uuid : 2.2.2 » a ajouté au fichier pubspec.yaml et les deux lignes du haut :

```
import 'package:uuid/uuid.dart';

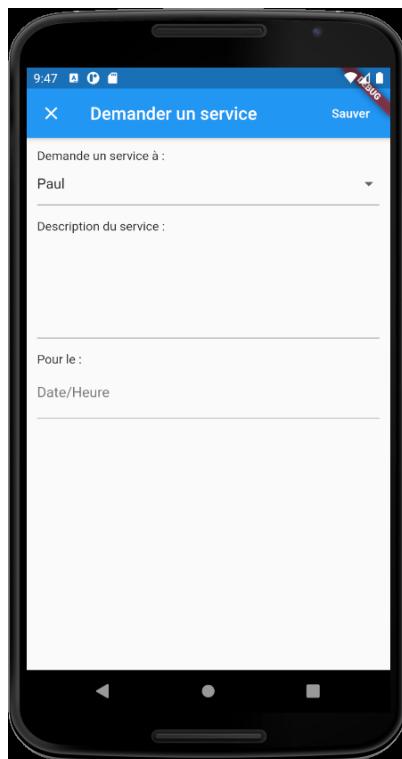
final uuid = Uuid();
```

permettent la génération de nombres aléatoires cryptographiquement forte sur toutes les plateformes en guise de clé.

6.1.5.3 La page de demande de service

6.1.5.3.1 La mise en page

L'écran de demande de service sera l'endroit d'interaction utilisateur-application. Pour l'instant, vous n'allez faire que la disposition de cet écran. Les interactions avec l'utilisateur ainsi que l'enregistrement du service dans la base de données distante (type Firebase) seront abordées dans les chapitres suivants.



Le widget d'écran de demande de service contient également un widget *Scaffold Material Design* avec une barre d'application qui contient des actions cette fois. Le corps du widget *Scaffold* contient les champs qui permettront la saisie des informations par l'utilisateur pour la création d'une demande de service.

6.1.5.3.2 Le code de la page

Le widget *PageDemandeService* est également un widget *Stateless* pour le moment car nous ne nous soucions que de sa mise en page pour le moment :

```
import 'package:flutter/material.dart';
import 'package:flutter/cupertino.dart';
import 'package:flutter/services.dart';
import 'package:intl/intl.dart';
import 'package:datetime_picker_formfield/datetime_picker_formfield.dart';

import 'amis.dart';

class PageDemandeService extends StatelessWidget {
    final List<Amis> amis;

    PageDemandeService({Key key, this.amis}) : super(key: key);

    @override
    Widget build(BuildContext context) {
        . .
    }
}
```

Comme vous pouvez le voir, le seul paramètre du widget est la liste d'amis, qui doit être fournie par le widget parent car il s'agit d'une instance *StatelessWidget* pour le moment. Les aspects de la navigation entre les écrans et pages seront abordés dans le chapitre 6.4.

La méthode *build ()* du widget commence par :

```
@override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text('Demander un service'),
            leading: CloseButton(),
            actions: <Widget>[
                FlatButton(
                    child: Text('Sauver'),
                    textColor: Colors.white,
                    onPressed: () {}),
            ],
        ),
    );
}
```

appBar contient ici deux nouvelles propriétés:

- **leading** : Cette propriété est un widget affiché avant le titre. Dans ce cas, vous utilisez un widget *CloseButton* qui est un bouton intégré au widget *Navigator* de *Material Design*.
- **actions** : Cette propriété reçoit une liste de widgets à afficher sous le titre. Dans ce cas, vous affichez une instance *FlatButton* à l'aide de laquelle vous enregistrerez la demande de faveur.

Le corps du widget *Scaffold* définit la disposition par un widget *Column*. Il contient deux nouvelles propriétés :

- **mainAxisSize** : Cette propriété définit la taille dans l'axe vertical. Ici, vous utilisez *MainAxisSize.min* afin qu'il ne prenne que l'espace nécessaire.
- **crossAxisAlignment** : Cette propriété définit où aligner les enfants sur l'axe horizontal. Par défaut, *Column* aligne ses enfants horizontalement au centre. En utilisant cette propriété, vous pouvez modifier ce comportement.

Il y a trois widgets enfants dans *Column* qui permettront d'effectuer la saisie de l'entrée de l'utilisateur :

- **DropdownButtonFormField** : C'est un widget qui répertorie les éléments du widget *DropdownMenuItem* dans une fenêtre contextuelle lorsque vous appuyez dessus :

```
DropdownButtonFormField(  
    onChanged: (_){},  
    items: amis  
        .map(  
            (a) => DropdownMenuItem(  
                child: Text(a.nom),  
            ),  
        )  
        .toList(),  
,
```

Ici, nous utilisons la méthode *map()* du type *Iterable*, où chaque élément de la liste (amis, dans ce cas) est mappé à un nouveau widget *DropdownMenuItem*. Ainsi, chaque élément de la liste d'amis sera affiché en tant qu'élément de widget dans la liste déroulante.

- **TextField** : Il permet la saisie du texte à partir du clavier virtuel :

```
TextField(  
    maxLines: 5,  
    inputFormatters: [  
        LengthLimitingTextInputFormatter(200),  
    ],  
,
```

En y ajoutant *inputFormatters*, vous pouvez configurer son apparence à l'écran. Ici, vous limitez simplement la longueur totale du texte tapé à 200 caractères en utilisant la classe *LengthLimitingTextInputFormatter*, qui est fournie par la bibliothèque « *flutter/services.dart* ».

- ⊕ **DateTimeField**: Il permet à l'utilisateur de sélectionner une instance de *DateTime* et de l'affecter à une variable de type *DateTime* :

```
DateTimeField(  
    format: DateFormat("dd/MM/yyyy 'à' HH:mm"),  
    decoration: InputDecoration(  
        labelText: 'Date/Heure',  
        floatingLabelBehavior: FloatingLabelBehavior.auto,  
    ),  
    onShowPicker: (context, currentValue) async {  
        final date = await showDatePicker(  
            context: context,  
            firstDate: DateTime(1900),  
            initialDate: currentValue ?? DateTime.now(),  
            lastDate: DateTime(2100));  
        if (date != null) {  
            final time = await showTimePicker(  
                context: context,  
                initialTime: TimeOfDay.fromDateTime(  
                    currentValue ?? DateTime.now()),  
                builder: (context, child) {  
                    return MediaQuery(  
                        data: MediaQuery.of(context)  
                            .copyWith(alwaysUse24HourFormat: true),  
                        child: child,  
                    );  
                },  
            );  
            return DateTimeField.combine(date, time);  
        } else {  
            return currentValue;  
        }  
    },  
    onChanged: (dt) {},  
,
```

Le widget *DateTimeField* n'est pas un widget intégré de Flutter. Il s'agit d'un plugin tiers de la bibliothèque *datetime_picker_formfield.dart*. Ici, vous définissez quelques propriétés pour changer son apparence :

format : cette propriété permet de définir le format d'affichage sous forme de chaîne de la valeur à partir d'une instance *DateFormat*.

decoration : cette propriété permet de définir une décoration pour le champ de saisie. Notez que nous ne l'avons pas défini pour les autres champs de saisie.

onShowDatePicker : permet d'afficher un calendrier pour choisir une date

onShowTimePicker : permet d'afficher une horloge pour choisir une heure

Dans ce dernier widget, vous forcez le format d'affichage en 24 heures en modifiant le *context* sur la propriété *alwaysUse24HourFormat*.

onChanged : appelé avec la nouvelle valeur sélectionnée par l'utilisateur.

Outre les champs de saisie, il existe également des widgets *Container* et *Text* dans la colonne pour aider à la mise en forme et à la conception de l'écran.

Les aspects de gestion des entrées et des gestes de l'utilisateur seront abordés dans le chapitre suivant.

6.2 Gestion des entrées et des gestes de l'utilisateur

Avec l'utilisation de widgets, il est possible de créer une interface riche en ressources visuelles qui permet également l'interaction de l'utilisateur par le biais de gestes et de saisie de données. Pour cela, vous découvrirez les widgets utilisés pour gérer les gestes de l'utilisateur, recevoir et valider les entrées de l'utilisateur, ainsi que comment créer nos propres entrées personnalisées.

6.2.1 Gestion des gestes de l'utilisateur

Une application mobile ne serait rien sans une sorte d'interactivité. Le framework Flutter permet de gérer les gestes de l'utilisateur de toutes les manières possibles, des simples tapotements aux gestes de glisser-déplacer. Les événements d'écran dans le système de gestes de Flutter sont séparés en deux couches :

Couche pointeur : c'est la couche qui gère des événements pointage représentant les interactions de l'utilisateur, avec des détails tels que l'emplacement tactile et le mouvement sur l'écran de l'appareil.

Couche geste : la couche gestes dans Flutter gère des événements d'interaction d'un plus haut niveau de définition. Vous en avez peut-être déjà vu certains en action, tels que les tapotements, les traînées et le zoom, par exemple. En outre, ils constituent le moyen le plus courant d'implémenter la gestion des événements.

6.2.1.1 Les pointeurs

Flutter démarre la gestion des événements dans une couche de bas niveau (couche pointeur), où vous pouvez gérer chaque événement de pointage et décider comment le contrôler, par exemple avec un glissement ou un simple tap.

Le framework Flutter implémente la distribution d'événements sur l'arborescence des widgets en suivant la séquence suivante:

- ⊕ *PointerDownEvent* est l'endroit où l'interaction commence, avec un appui entrant en contact avec un certain emplacement de l'écran de l'appareil. Ici, le framework recherche dans l'arborescence des widgets le widget qui existe à l'emplacement du pointeur sur l'écran. Cette action s'appelle un test de réussite.
- ⊕ Chaque événement suivant est distribué au widget le plus bas qui correspond à l'emplacement, puis élevé dans l'arborescence des widgets vers les widgets parents jusqu'à la racine. Cette propagation des actions événementielles ne peut pas être interrompue. L'événement peut être *PointerMoveEvent*, où l'emplacement du pointeur est modifié. Il peut également s'agir de *PointerUpEvent* ou *PointerCancelEvent*.
- ⊕ Une interaction peut se terminer par *PointerUpEvent* ou *PointerCancelEvent*. Le premier ici est l'endroit où le pointeur cesse d'être en contact avec l'écran, tandis que le second signifie que l'application ne reçoit plus d'événements sur le pointeur (l'événement n'est pas complet).

Flutter fournit la classe *Listener*, qui peut être utilisée pour détecter les événements d'interaction de pointeur. Vous pouvez l'intégrer à une arborescence de widgets, alors dans ce cas, ce widget pourra gérer les événements de pointeur sur son sous-arbre.

6.2.1.2 Les gestes

Bien que possible, il n'est pas toujours pratique de gérer nous-mêmes les événements de pointeur à l'aide du widget *Listener*. Au lieu de cela, les événements peuvent être gérés sur la deuxième couche du système de gestes Flutter. Les gestes sont reconnus à partir de plusieurs événements de pointeur, et même de plusieurs pointeurs individuels (*multitouch*).

Il existe plusieurs types de gestes qui peuvent être gérés :

- **Simple Clic/Appui** : Un simple appui sur l'écran de l'appareil.
- **Double Clic/Appui** : Un appui double rapide sur le même emplacement sur l'écran de l'appareil.
- **Clic/Appui long** : Un appui pendant une longue période sur l'écran.
- **Glisser** : Un appui qui commence à un endroit de l'écran et relâché à un autre endroit sur l'écran de l'appareil.
- **Déplacer** : similaire aux événements de glissement. Les gestes de déplacement ont une direction quelconque.
- **Mettre à l'échelle** : deux appuis simultanés utilisés pour un mouvement de glissement pour créer un geste de mise à l'échelle. Ceci est également similaire à un geste de zoom.

Comme le widget *Listener* pour les événements de pointeur, Flutter fournit le widget *GestureDetector*, qui contient les appels pour tous les événements précédents. Nous devons les utiliser en fonction de l'effet que nous voulons obtenir.

6.2.1.2.1 Simple Clic/Appui

Voyez comment implémenter l'événement à l'aide de l'appel *onTap* du widget *GestureDetector*:

```
class _TapWidgetExampleState extends State<TapWidgetExample> {
    int _counter = 0;

    @override
    Widget build(BuildContext context) {
        return GestureDetector(
            onTap: () {
                setState(() {
                    _counter++;
                });
            },
            child: Container(
                color: Colors.grey,
                child: Center(
                    child: Text(
                        "Tap count: $_counter",
                        style: Theme.of(context).textTheme.headline4,
                    ),
                ),
            ),
        );
    }
}
```

Il s'agit de l'implémentation de l'état d'un widget. Ici dans l'exemple, l'événement *onTap* permet d'incrémenter un compteur *_counter* pour montrer combien d'appui ont été effectués sur l'écran.

6.2.1.2.2 Double Clic/Appui

L'appel de double tap est très similaire dans le code:

```

class _DoubleTapWidgetExampleState extends State<DoubleTapWidgetExample> {
    int _counter = 0;

    @override
    Widget build(BuildContext context) {
        return GestureDetector(
            onDoubleTap: () {
                setState(() {
                    _counter++;
                });
            },
            child: Container(
                color: Colors.grey,
                child: Center(
                    child: Text(
                        "DoubleTap count: $_counter",
                        style: Theme.of(context).textTheme.headline4,
                    ),
                ),
            ),
        );
    }
}

```

La seule différence par rapport à l'exemple précédent est la propriété attribuée, *onDoubleTap*, qui sera appelée chaque fois que des doubles clics/appuis sont rapidement effectués au même endroit sur l'écran.

6.2.1.2.3 Clic/Appui long

La différence par rapport aux exemples précédents est minime :

```

class _PressAndHoldWidgetExampleState extends State<PressAndHoldWidgetExample>
{
    int _counter = 0;

    @override
    Widget build(BuildContext context) {
        return GestureDetector(
            onLongPress: () {
                setState(() {
                    _counter++;
                });
            },
            child: Container(
                color: Colors.grey,
                child: Center(
                    child: Text(
                        "Long press count: $_counter",
                        style: Theme.of(context).textTheme.headline4,
                    ),
                ),
            ),
        );
    }
}

```

```

        ),
        ),
        );
    }
}

```

La seule différence par rapport à l'élément précédent est la propriété attribuée, *onLongPress*, qui sera appelée chaque fois qu'un appui est effectué et maintenu pendant un certain temps - un appui long - avant d'être libéré de l'écran.

6.2.1.2.4 Glisser

Les mouvements de glisser, de déplacer et de mise à l'échelle sont similaires, vous devez décider lequel utiliser dans chaque situation, car ils ne peuvent pas être utilisés tous ensemble dans le même widget *GestureDetector*.

Les gestes de glissement sont séparés en gestes verticaux et horizontaux. Même les rappels sont séparés dans Flutter.

6.2.1.2.4.1 Glissement horizontal

Le code ci-dessous vous donne un exemple de glisser horizontal :

```

class _HorizontalDragWidgetExampleState
    extends State<HorizontalDragWidgetExample> {
    bool _dragging = false;
    Offset _move = Offset.zero;
    int _dragCount = 0;

    @override
    Widget build(BuildContext context) {
        return GestureDetector(
            onHorizontalDragStart: (DragStartDetails details) {
                setState(() {
                    _move = Offset.zero;
                    _dragging = true;
                });
            },
            onHorizontalDragUpdate: (DragUpdateDetails details) {
                setState(() {
                    _move += details.delta;
                });
            },
            onHorizontalDragEnd: (DragEndDetails details) {
                setState(() {
                    _dragging = false;
                    _dragCount++;
                });
            },
            child: Container(

```

```

        color: Colors.grey,
        child: Center(
            child: Transform.translate(
                offset: _move,
                child: Text(_dragging ? "DRAGGING!" : "Drags: ${_dragCount}",
                    style: Theme.of(context).textTheme.headline4),
            ),
        ),
    ),
);
}
}

```

Cette fois, vous avez un peu plus de travail que pour les événements *tap*. Dans l'exemple, vous avez trois propriétés présentes dans l'état :

_dragging: Utilisé pour mettre à jour le texte visualisé par l'utilisateur lors du glissement.

_dragCount: Cela accumule le nombre total d'événements de glissement effectués.

_move: Cela accumule le décalage du glissement appliqué au texte à l'aide du constructeur de traduction du widget *Transformer*.

Comme vous pouvez le voir, les appels de glissement reçoivent des paramètres liés à chaque événement - *DragStartDetails*, *DragUpdateDetails* et *DragEndDetails* - qui contiennent des valeurs qui peuvent aider à chaque étape du glissement.

6.2.1.2.4.2 Glissement vertical

Le code du glissement vertical est presque le même que le code du glissement horizontal.

```

class _VerticalDragWidgetExampleState extends State<VerticalDragWidgetExample>
{
    bool _dragging = false;
    Offset _move = Offset.zero;
    int _dragCount = 0;

    @override
    Widget build(BuildContext context) {
        return GestureDetector(
            onVerticalDragStart: (DragStartDetails details) {
                setState(() {
                    _move = Offset.zero;
                    _dragging = true;
                });
            },
            onVerticalDragUpdate: (DragUpdateDetails details) {
                setState(() {
                    _move += details.delta;
                });
            },
        );
    }
}

```

```
onVerticalDragEnd: (DragEndDetails details) {
    setState(() {
        _dragging = false;
        _dragCount++;
    });
},
child: Container(
    color: Colors.grey,
    child: Center(
        child: Transform.translate(
            offset: _move,
            child: Text(
                _dragging ? "DRAGGING!" : "Drags: ${_dragCount}",
                style: Theme.of(context).textTheme.headline4,
            ),
        ),
    ),
),
),
),
);
}
}
```

Les différences significatives résident dans les propriétés des appels, qui sont *onVerticalDragStart*, *onVerticalDragUpdate* et *onVerticalDragEnd*.

6.2.1.2.5 Déplacer

Le code du déplacer est également très similaire.

```
class _PanWidgetExampleState extends State<PanWidgetExample> {
  bool _dragging = false;
  Offset _move = Offset.zero;
  int _dragCount = 0;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onPanStart: (DragStartDetails details) {
        setState(() {
          _move = Offset.zero;
          _dragging = true;
        });
      },
      onPanUpdate: (DragUpdateDetails details) {
        setState(() {
          _move += details.delta;
        });
      },
      onPanEnd: (DragEndDetails details) {
        setState(() {
```

```
        _dragging = false;
        _dragCount++;
    });
},
child: Container(
    color: Colors.grey,
    child: Center(
        child: Transform.translate(
            offset: _move,
            child: Text(
                _dragging ? "DRAGGING!" : "Drags: ${_dragCount}",
                style: Theme.of(context).textTheme.headline4,
            ),
        ),
    ),
),
);
}
}
```

Les différences significatives cette fois-ci s'ajoutent aux propriétés des appels, qui sont désormais `onPanStart`, `onPanUpdate` et `onPanEnd`. Pour les déplacements, les deux décalages d'axe sont évalués. Autrement dit, les deux valeurs delta dans `DragUpdateDetails` sont présentes, de sorte que le glissement n'a aucune contrainte de direction.

6.2.1.2.6 Mettre à l'échelle

Le code de mise à l'échelle ou zoom n'est rien d'autre qu'un déplacement sur plus d'un pointeur (voir code ci-dessous).

```
class _ScaleWidgetExampleState extends State<ScaleWidgetExample> {
  bool _resizing = false;
  double _scale = 1.0;
  int _scaleCount = 0;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onScaleStart: (ScaleStartDetails details) {
        setState(() {
          _scale = 1.0;
          _resizing = true;
        });
      },
      onScaleUpdate: (ScaleUpdateDetails details) {
        setState(() {
          _scale = details.scale;
        });
      },
      onScaleEnd: (ScaleEndDetails details) {
        setState(() {
          _scale = 1.0;
        });
      },
    );
  }
}
```

```

        setState(() {
            _resizing = false;
            _scaleCount++;
        });
    },
    child: Container(
        color: Colors.grey,
        child: Center(
            child: Transform.scale(
                scale: _scale,
                child: Text(
                    _resizing ? "RESIZING!" : "Scale count: ${_scaleCount}",
                    style: Theme.of(context).textTheme.headline4,
                ),
            ),
        ),
    ),
);
}
}

```

Vous avez trois propriétés dans l'état :

`_resizing`: Celle-ci est utilisé pour mettre à jour le texte visualisé par l'utilisateur lors du redimensionnement à l'aide du geste de mise à l'échelle.

`_scaleCount`: Celle-ci accumule le nombre total d'événements réalisés.

`_scale`: Celle-ci stocke la valeur d'échelle du paramètre `ScaleUpdateDetails`, et elle est appliquée ultérieurement au widget `Text` à l'aide du constructeur d'échelle du widget `Transform`.

Comme vous pouvez le voir, les appels de mise à l'échelle ressemblent beaucoup aux appels de glissement en ce sens qu'ils reçoivent également des paramètres liés à chaque événement - `ScaleStartDetails`, `ScaleUpdateDetails` et `ScaleEndDetails` - qui contiennent des valeurs qui peuvent aider à chaque étape de l'événement.

6.2.1.2.7 Gestures par rapport aux plateformes

Les widgets *Material Design* et *iOS Cupertino* ont de nombreux gestes particuliers pour certaines propriétés lors de l'utilisation du widget `GestureDetector`.

Par exemple, les widgets de *Material Design* tels que `RaisedButton` utilisent le widget `InkWell` à côté de l'événement `tap`. Il provoque un effet d'éclaboussure sur le widget cible. En outre, la propriété `onPressed` de `RaisedButton` expose la fonctionnalité de `tap` qui peut être utilisée pour implémenter l'action du bouton.

6.2.2 Les widgets d'entrée

La gestion des gestes est un bon point de départ pour l'interaction avec l'utilisateur, mais ce n'est évidemment pas suffisant. L'obtention de données utilisateur est ce qui ajoute du contenu à de nombreuses applications.

Flutter fournit de nombreux widgets de saisie de données pour aider le développeur à obtenir différents types d'informations de l'utilisateur. Vous en avez déjà vu quelques-uns dans les chapitres précédents comme *TextField*, et différents types de widgets *Selector* et *Picker*.

Bien que vous puissiez gérer toutes les données saisies par l'utilisateur par vous-mêmes (disons, dans un widget racine qui contient tous les champs de saisie), cela peut devenir fastidieux, car cela pourrait vous conduire à avoir de nombreux champs et vous finiriez donc par obtenir un code de complexité croissante. Diviser tous les widgets d'entrée en petits morceaux aide, mais ne résout pas tout.

Flutter fournit deux widgets pour aider à organiser la saisie dans le code, la valider et fournir rapidement des commentaires à l'utilisateur. Ce sont les widgets *FormField* et *Form*.

6.2.2.1 Le widget *FormField*

Le widget *FormField* fonctionne comme une classe de base pour créer votre propre champ de formulaire en intégrant le widget *Form*. Ses fonctions sont les suivantes :

- ⊕ Aider dans le processus de paramétrage et de récupération de la valeur saisie.
- ⊕ Valider la valeur saisie.
- ⊕ Fournir un retour à l'utilisateur à partir des validations.

FormField peut vivre sans widgets *Form*, mais ce n'est pas typique - uniquement lorsque vous avez, disons, un seul *FormField* à l'écran. De nombreux widgets d'entrée intégrés de Flutter sont fournis avec une implémentation du widget *FormField* correspondante. Par exemple, le widget *TextField* a le *TextFormField*.

Le widget *TextFormField* facilite l'accès à la valeur *TextField* et lui ajoute également des comportements liés au formulaire (tels que la validation). Un widget *TextField* permet à l'utilisateur de saisir du texte au clavier. Le widget *TextField* expose la méthode *onChanged*, qui peut être utilisée pour obtenir les modifications de sa valeur actuelle. Une autre façon d'obtenir les changements consiste à utiliser d'un contrôleur.

6.2.2.2 Utilisation d'un contrôleur

Lorsqu'il est isolé hors d'un formulaire, c'est-à-dire en utilisant uniquement le widget *TextField*, nous devons utiliser sa propriété *controller* pour accéder à sa valeur. Cela se fait avec la classe *TextEditingController* :

```
final _controller = TextEditingController.fromValue(  
    TextEditingValue(text: "Initial value"),  
)
```

Après avoir instancié le *TextEditingController*, nous le définissons dans la propriété *controller* du widget *TextField* afin qu'il "contrôle" le widget texte:

```
child: TextField(  
    controller: _controller,  
)
```

Comme vous pouvez le voir, nous pouvons également définir une valeur initiale pour *TextField*. *TextEditingController* est notifié chaque fois que le widget *TextField* a une nouvelle valeur. Pour intercepter les changements, nous devons ajouter une méthode *addListener* à notre *_controller*.

```
_controller.addListener(_textFieldEvent);
```

_textFieldEvent doit être une fonction qui sera appelée à chaque fois que le widget *TextField* change.

6.2.2.3 Accéder à l'état de *FormField*

Si nous utilisons le widget *TextFormField*, les choses deviennent plus simples :

```
final _key = GlobalKey<FormFieldState<String>>();  
...  
TextFormField(  
    key: _key,  
);
```

Nous pouvons ajouter une clé *_key* à notre *TextFormField* qui pourra être utilisée ultérieurement pour accéder à l'état actuel du widget via la valeur *_key.currentState*, qui contiendra la valeur mise à jour du champ.

Le type de *_key* fait référence au type de données avec lesquelles le champ de saisie fonctionne. Dans l'exemple précédent, il s'agit de *String*, car il s'agit d'un widget *TextField*.

La classe *FormFieldState <String>* fournit également d'autres méthodes et propriétés utiles pour traiter *FormField*:

- ⊕ **validate()** appellera la fonction de validation de la saisie du widget, qui devrait vérifier sa valeur actuelle et renvoyer un message d'erreur, ou *null* s'il est valide.
- ⊕ **hasError** et **errorText** donnera le résultat de validations précédentes à l'aide de la fonction de validation précédente. Dans les widgets Material Design, par exemple, cela ajoute un petit texte près du champ, fournissant à l'utilisateur une rétroaction appropriée sur l'erreur.
- ⊕ **save()** appellera la fonction correspondant à *onSaved* du widget. Il s'agit de l'action qui se produit lorsque la saisie est effectuée par l'utilisateur (lors de son enregistrement).
- ⊕ **reset()** mettra le champ dans son état initial, avec la valeur initiale (le cas échéant), effaçant également les erreurs de validation.

6.2.2.4 Le widget *Form*

Le fait d'avoir un *FormFieldWidget* nous aide à accéder et à valider individuellement ses informations. Mais, pour résoudre le problème d'avoir trop de champs, nous pouvons utiliser le widget *Form*.

Le widget *Form* regroupe les instances *FormFieldWidget* de manière logique, ce qui nous permet d'effectuer des opérations, notamment d'accéder aux informations de champ et de les valider de manière plus simple.

Le widget *Form* nous permet d'exécuter facilement les méthodes suivantes sur tous les champs enfants :

- ⊕ **save()**: Cela appellera la méthode de sauvegarde de toutes les instances *FormField*. C'est comme une sauvegarde par lots de tous les champs.
- ⊕ **validate()**: Cela appellera la méthode de validation de toutes les instances *FormField*, provoquant l'apparition de toutes les erreurs en même temps.

- ➡ **reset():** Cela appellera la méthode de réinitialisation de toutes les instances *FormField*. Cela amènera le formulaire entier à son état initial.

6.2.2.5 Accès à l'état du widget *Form*

Fournir l'accès à l'objet actuel associé à l'état du widget *Form* est utile pour que nous puissions lancer sa validation, enregistrer son contenu ou le réinitialiser de n'importe où dans l'arborescence des widgets (c'est-à-dire en appuyant sur un bouton). Il existe deux façons d'accéder à l'état associé au widget *Form*.

6.2.2.5.1 Utilisation d'une clé *key*

Le widget *Form* est utilisé conjointement avec une clé de type *FormState*, qui contient des *helpers* pour gérer tous les enfants de ses instances *FormField*:

```
/// Accessing field state by using a key

class InputFormFieldExamplesWidget extends StatelessWidget {
  final _key = GlobalKey<FormFieldState<String>>();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: GestureDetector(
        onTap: () {
          print("Current value ${_key.currentState.value}");
        },
        child: Container(
          color: Colors.grey,
          child: Center(
            child: TextFormField(
              key: _key,
            ),
          ),
        ),
      ),
    );
  }
}
```

Ensuite, nous pouvons utiliser la clé pour récupérer l'état associé au widget *Form* et appeler sa validation avec *_key.currentState.validate()*.

6.2.2.5.2 Utilisation *InheritedWidget*

Le widget *Form* est livré avec une classe utile pour se passer de la nécessité d'ajouter une clé et de toujours bénéficier de ses avantages.

Chaque widget *Form* de l'arborescence est associé à un *InheritedWidget*. *Form* et de nombreux autres widgets exposent cela dans une méthode statique appelée *of()*, où nous passons *BuildContext*. Il recherche dans l'arborescence pour trouver l'état correspondant que nous recherchons. Sachant cela, si nous devons accéder au widget *Form* quelque part en dessous dans l'arborescence, nous pouvons

utiliser `Form.of()`, et nous avons accès aux mêmes fonctions que nous aurions si nous utilisons la propriété `key`.

```
/// Accessing form by InheritedWidget, click some part of body
class InputFormInheritedStateExamplesWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        color: Colors.grey,
        alignment: Alignment.center,
        child: Form(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.min,
            children: <Widget>[
              TextFormField(
                validator: (String value) {
                  return value.isEmpty ? "cannot be empty" : null;
                },
              ),
              Builder(
                builder: (BuildContext context) => RaisedButton(
                  onPressed: () {
                    print("Running validation");
                    final valid = Form.of(context).validate();
                    print("valid: $valid");
                  },
                  child: Text("validate"),
                ),
              ),
            ],
          ),
        ),
      );
    }
}
```

Portez une attention particulière au widget `Builder` utilisé pour retourner `RaisedButton`. Comme nous l'avons vu précédemment, le widget hérité peut être regardé dans l'arbre.

Considérez l'utilisation suivante de `RaisedButton` directement dans le widget `Column` :

```
child: Column(
  children: <Widget>[
    TextFormField(),
    RaisedButton(
      onPressed: () {
        print("Running validation");
      },
    ),
  ],
);
```

```
    final valid = Form.of(context).validate(); // Wrong context
    print("valid: $valid");
},
child: Text("validate"),
),
],
),
);
```

Lorsque nous utilisons `Form.of(context)`, nous transmettons le contexte actuel du widget. Dans l'exemple précédent, le contexte utilisé dans l'appel `onPressed` sera le contexte `InputFormInheritedStateExamplesWidget`, et ainsi, la recherche de l'arborescence ne trouvera pas avec succès un widget `Form`.

En utilisant le widget *Builder*, nous déléguons sa construction à un appel, cette fois en utilisant le contexte correct (celui de l'enfant), et quand il recherche dans l'arborescence, il trouvera avec succès l'instance *FormState*.

6.2.2.6 Validation des entrées : widget Form

La gestion de plusieurs widgets *FormField* est acceptable lorsque l'on parle de peu de valeurs, mais lorsque la quantité de données augmente, l'organisation à l'écran, la validation de tout et la communication rapide des commentaires vers les utilisateurs peuvent devenir plus difficiles. C'est pourquoi Flutter fournit le widget *Form*.

6.2.2.6.1 Validation d'une saisie utilisateur

La validation de la saisie utilisateur est l'une des principales fonctions du widget Form. Afin de rendre cohérente la saisie de données par l'utilisateur, il est fondamental de la vérifier, car l'utilisateur ne connaît probablement pas toutes les valeurs autorisées. Le widget *Form*, combiné avec des instances *FormField*, aide le développeur à afficher un message d'erreur approprié si certaines valeurs d'entrée doivent être corrigées avant d'enregistrer les données du widget *Form* via sa fonction *save()*. Nous avons déjà vu, dans les exemples précédents de *Form* comment valider les valeurs du champ *Form*:

- a) Créez un widget *Form* avec un *FormField* au-dessus.
 - b) Définir la logique de validation sur chaque propriété *validator* du *FormField*:
 - c) Appeler *validate()* sur *FormState* en utilisant sa clé, ou la méthode *Form.of*. Cela appellera chaque méthode enfant *FormField validate()*, et lorsque la validation réussit, elle retournera *true* et *false* dans le cas contraire.
 - d) *validate()* retourne un booléen afin que nous puissions manipuler son résultat.

```
    TextFormField(  
        validator: (String value) {  
            return value.isEmpty ? "cannot be empty" : null;  
        },  
    ),
```

Nous avons vu comment les widgets *Form* et *FormField* facilitent la manipulation et la validation des entrées. En outre, nous savons que Flutter est livré avec une série de widgets d'entrée qui sont des variantes de *FormField*, et contiennent donc des fonctions d'assistance pour accéder et valider les données. L'extensibilité et la flexibilité de Flutter sont partout. Ainsi, la création de champs personnalisés est logiquement possible, où nous pouvons ajouter notre propre méthode d'entrée, exposer la validation via l'appel du *validator*, et également utiliser les méthodes *save()* et *reset()*.

6.2.3 Création d'entrées personnalisées

La création d'une entrée personnalisée dans Flutter est aussi simple que la création d'un widget normal, en intégrant les méthodes supplémentaires décrites précédemment. Nous le faisons normalement en étendant le widget *FormField<inputType>*, où *inputType* est le type de valeur du widget d'entrée.

Ainsi, le processus est le suivant :

- a) Créez un widget personnalisé qui étend le widget *Stateful* (pour garder une trace de la valeur) et accepte l'entrée de l'utilisateur en encapsulant un autre widget d'entrée, ou en personnalisant l'ensemble du processus, par exemple en utilisant des gestes.
- b) Créez un widget qui étend *FormField* qui affiche essentiellement le widget d'entrée créé à l'étape précédente et expose également ses champs.

6.2.3.1 Exemple de widget d'entrée personnalisé

Plus tard, dans l'utilisation de Firebase, nous verrons comment ajouter une authentification à notre application. Pour l'instant, nous allons créer un widget personnalisé qui sera similaire à celui utilisé à cette étape. L'authentification sera basée sur les services d'authentification Firebase qui utilisent un numéro de téléphone. Le numéro de téléphone fourni reçoit un code de vérification à six chiffres qui doit correspondre à la valeur du serveur pour pouvoir se connecter. Pour l'instant, ce sont toutes les informations dont nous avons besoin pour créer le widget d'entrée personnalisé (voir écran ci-dessous).



Le widget sera un simple widget d'entrée à six chiffres, qui deviendra plus tard un widget *FormField* et exposera les méthodes *save()*, *reset()* et *validate()*.

6.2.3.2 Création du widget d'entrée de base

Nous commençons par créer un widget personnalisé. Ici, nous exposons quelques propriétés. Gardez à l'esprit que dans une application réelle, nous exposerions probablement plus que les propriétés exposées ici, mais cela suffit pour cet exemple :

```
class VerificationCodeInput extends StatefulWidget {
    final BorderSide borderSide;
    final onChanged;
    final controller;

    const VerificationCodeInput({
        Key key,
        this.controller,
        this.borderSide = const BorderSide(),
        this.onChanged,
    }) : super(key: key);

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

La seule propriété importante exposée ici est *controller*. Nous verrons la raison dans quelques instants. Commençons par vérifier la classe *State* associée :

```
class _VerificationCodeInputState extends State<VerificationCodeInput> {
    @override
    Widget build(BuildContext context) {
        return TextField(
            controller: widget.controller,
            inputFormatters: <TextInputFormatter>[
                FilteringTextInputFormatter.allow(RegExp("[0-9]")),
                LengthLimitingTextInputFormatter(6),
            ],
            textAlign: TextAlign.center,
            decoration: InputDecoration(
                border: OutlineInputBorder(
                    borderSide: widget.borderSide,
                ),
            ),
            keyboardType: TextInputType.number,
            onChanged: widget.onChanged,
        );
    }
}
```

Comme vous pouvez le voir, le widget est simplement un *TextField* avec une personnalisation prédefinie :

- FilteringTextInputFormatter nous permet de spécifier une expression *regex* avec les caractères autorisés pour l'entrée. En définissant le type de clavier avec keyboardType: TextInputType.number, nous pouvons également limiter les caractères autorisés aux nombres.
- LengthLimitingTextInputFormatter spécifie une limite maximale de caractères pour l'entrée.
- Une bordure est ajoutée via la classe OutlineInputBorder.

Prenez note de la partie importante de ce code : controller: widget.controller. Ici, nous définissons le contrôleur du widget *TextField* pour qu'il soit notre contrôleur afin que nous puissions prendre le contrôle de sa valeur.

6.2.3.3 Transformez le widget en widget *FormField*

Pour transformer le widget en widget *FormField*, nous commençons par créer un widget qui étend la classe *FormField*, qui est un *StatefulWidget* avec quelques fonctionnalités *Form*. Cette fois, commençons par vérifier l'objet *State* associé au nouveau widget.

```
class _VerificationCodeFormFieldState extends FormFieldState<String> {
  final TextEditingController _controller = TextEditingController(text: "");

  @override
  void initState() {
    super.initState();

    _controller.addListener(_controllerChanged);
  }

  @override
  void reset() {
    super.reset();
    _controller.text = "";
  }

  void _controllerChanged() {
    didChange(_controller.text);
  }

  @override
  void dispose() {
    _controller?.removeListener(_controllerChanged);
    super.dispose();
  }
}
```

À partir du code précédent, vous pouvez vérifier qu'il contient un seul champ *_controller*, qui représente le contrôleur utilisé par le widget *FormField*. Il doit être dans *State* afin qu'il persiste contre les changements de mise en page. Comme vous pouvez le voir, il est initialisé dans la fonction *initState()*. Celle-ci est appellée la première fois que l'objet widget est inséré dans l'arborescence des widgets. Ici, nous y ajoutons un *Listener*, afin que nous puissions savoir quand la valeur est modifiée dans l'écouteur *_controllerChanged*.

Il y a aussi d'autres méthodes importantes que nous devons redéfinir pour que cela fonctionne correctement :

- ⊕ **`initState()`**, nous pouvons trouver son opposé dans la méthode **`dispose()`**. Ici, nous nous arrêtons pour écouter les changements dans le contrôleur.
- ⊕ **`reset()`** est surchargée, pour pouvoir effacer le contenu `_controller.text`, rendant le champ de saisie à nouveau vide.
- ⊕ **`_controllerChanged()`** notifie l'état du *FormFieldState* via sa méthode `didChange()`, afin qu'il puisse mettre à jour son état (via `setState()`) et notifier le changement à tout widget *Form* qui le contient.

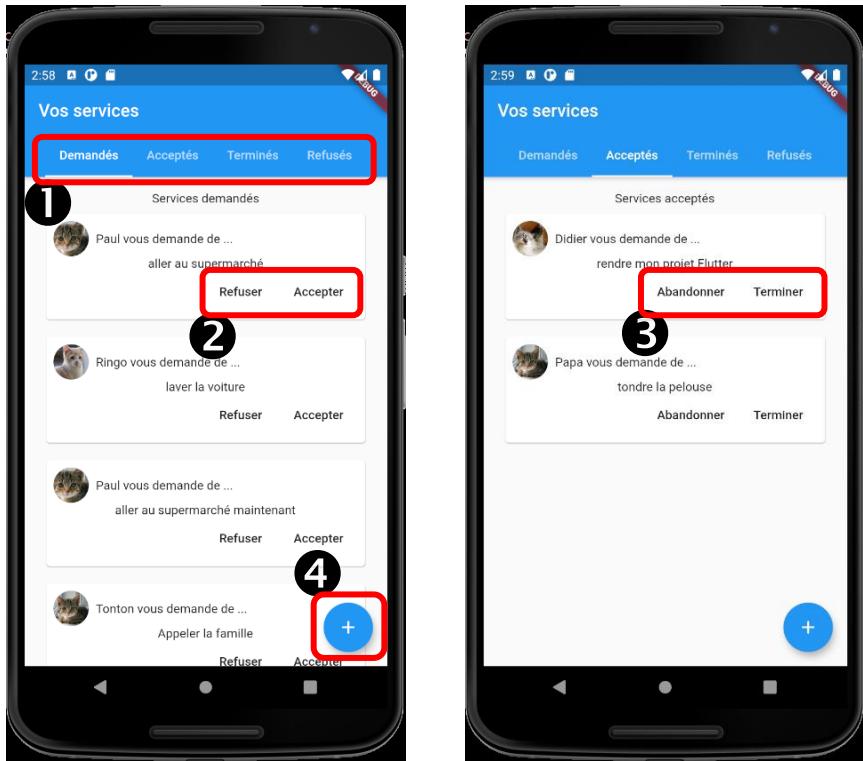
Examinons maintenant le code du widget *FormField* pour voir comment il fonctionne :

```
class VerificationCodeFormField extends FormField<String> {  
    final TextEditingController controller;  
  
    VerificationCodeFormField({  
        Key key,  
        FormFieldSetter<String> onSaved,  
        this.controller,  
        FormFieldValidator<String> validator,  
    }) : super(  
        key: key,  
        validator: validator,  
        builder: (FormFieldState<String> field) {  
            _VerificationCodeFormFieldState state = field;  
            return VerificationCodeInput(  
                controller: state._controller,  
            );  
        },  
    );  
  
    @override  
    FormFieldState<String> createState() => _VerificationCodeFormFieldState();  
}
```

La nouveauté ici est dans le constructeur. Le widget *FormField* contient l'appel à la méthode qui doit générer le widget d'entrée associé. Il transmet l'état actuel de l'objet afin que nous puissions créer le widget et conserver les informations actuelles. Comme vous pouvez le voir, nous l'utilisons pour transmettre le contrôleur `controller` construit dans l'état, de sorte qu'il persiste même lorsque le champ est reconstruit. C'est ainsi que nous maintenons le widget et l'état synchronisés, et intégrons également avec la classe *Form*.

6.2.4 Ma deuxième application flutter (suite)

A partir de l'application de gestion de services entre amis, nous allons voir comment utiliser les événements gestuels et les widgets d'entrée pour ajouter une interaction utilisateur à nos écrans d'application, il est temps de compléter notre application avec ces fonctions. Revoyons nos écrans pour y ajouter les gestes et les validations de saisie.



① Concentrez-vous sur la section de catégorie de service sélectionnée. Ceci est déjà fait pour nous par le widget *DefaultTabController* (il existe un widget *ListView* interne qui gérera les gestes de balayage et/ou défilement).

② « Refuser » ou « Accepter » les services demandés. Par exemple, un service a été demandé par un ami et l'utilisateur peut l'accepter ou le rejeter. Ainsi, en appuyant sur l'un des boutons, le service change son statut en *Refusés* ou *Acceptés*.

③ De même que dans le cas précédent, mais cette fois, une demande de service accepté est en attente d'achèvement, et ces boutons permettent à l'utilisateur d'abandonner ou de terminer un service rendu. C'est-à-dire qu'en appuyant dessus, le statut du service passe respectivement à *Refusés* et *Terminés*.

④ Nous avons le bouton « **Demandeur un service** », qui ouvre essentiellement un deuxième écran d'application lorsque vous appuyez dessus, ce qui nous permet de demander un service à certains de nos amis.

Comme vous pouvez le voir à partir des gestes précédents, nous traiterons des gestes de sélection (clic), de défilement et de balayage. Tous peuvent être effectués directement avec *GestureDetector*, mais comme nous utilisons les widgets *Button* et *ListView*. N'oubliez pas que les widgets intégrés de Flutter sont également composés de nombreux autres widgets intégrés, nous traiterons donc indirectement comme *GestureDetector*.

En pratique, nous gérerons les appuis simples par nous-mêmes, car les autres gestes sont gérés par les widgets que nous avons utilisés : le défilement avec *ListView*, et les balayages et les sélections avec *TabBar* et *TabView*.

6.2.4.1 Appui sur l'onglet du type de service

Comme nous l'avons vu précédemment, le *DefaultTabController* modifie le widget de l'onglet actuellement visible lorsque l'utilisateur appuie sur la barre d'onglets ou glisse vers la gauche ou la

droite sur la vue. En utilisant ce widget, nous n'avons pas besoin de spécifier un contrôleur dans les widgets descendants *TabBar* et *TabView*.

6.2.4.2 Appui sur un service

À partir de la propriété *service* du widget *_ElementListeServices*, nous pouvons manipuler son statut en modifiant ses valeurs de champ *accepte* et *complete*. Cependant, il ne supprimera pas l'élément de la liste actuelle et ne l'ajoutera pas à la nouvelle liste cible. Pour ce faire, nous aurions besoin d'accéder à la liste actuelle, de supprimer l'élément *service* à partir de là et de l'ajouter dans la nouvelle liste, en fonction du bouton enfoncé. Nous pourrions utiliser notre liste globale de services directement dans la méthode *onPressed* de l'élément, mais cela impliquerait de distribuer la logique métier via les widgets, ce qui semble bien maintenant, mais peut facilement devenir compliqué.

Alors, où devrions-nous gérer cette action efficacement ? Nous pourrions gérer toutes ces actions dans le widget *PageServices*, qui contient toutes les listes de faveurs. Mais attendez - *PageServices* est un *StatelessWidget*, les listes de faveurs sont chargées dans sa méthode *build*, et comme il est sans état, elles seront chargées à chaque reconstruction du widget, perdant nos modifications.

6.2.4.2.1 Faire de *PageServices* un *StatefulWidget*

La première étape pour rendre notre application interactive consiste à changer *PageServices* en *StatefulWidget* :

```
class PageServices extends StatefulWidget {
    PageServices({
        Key key,
    }) : super(key: key);
    @override
    _PageServicesState createState() => _PageServicesState();
}
```

La première chose que nous changeons est l'ancien widget *PageServices*, et maintenant son seul travail est de renvoyer une instance *_PageServicesState* dans la méthode *createState()* :

```
class _PageServicesState extends State<PageServices> {
    List<Service> servicesDemandes;
    List<Service> servicesTermimes;
    List<Service> servicesRefuses;
    List<Service> servicesAcceptes;

    @override
    void initState() {
        servicesDemandes = List();
        servicesTermimes = List();
        servicesRefuses = List();
        servicesAcceptes = List();

        chargerServices();

        super.initState();
    }
}
```

```

void chargerServices() {
    servicesDemandes.addAll(simulServicesDemandes);
    servicesTermimes.addAll(simulServicesTermimes);
    servicesRefuses.addAll(simulServicesRefuses);
    servicesAcceptes.addAll(simulServicesAcceptes);
}
@Override
Widget build(BuildContext context) { ... }
}

```

Maintenant, l'objet *State* contient les informations qui doivent être conservées entre les reconstructions, et cet objet sera l'emplacement de toutes les actions pour les services.

Alors, commençons par gérer les actions de demande en attente. Ils ont été définis comme *Refusés* ou *Acceptés*. Pour les gérer, nous devons passer un gestionnaire à la propriété *onPressed* de nos widgets *FlatButton* déjà définis dans le *_ElementListeServices*. À partir de la méthode *onPressed* du bouton, nous devons accéder d'une manière ou d'une autre à *_PageServicesState* pour effectuer ces actions. Cela peut être fait avec la méthode *findAncestorStateOfType ()* de la classe *BuildContext*, qui recherche dans l'arborescence un objet *State* du type donné :

```

static _PageServicesState of(BuildContext context) {
    return context.findAncestorStateOfType<_PageServicesState>();
}

```

Un modèle courant pour fournir cette fonction consiste à ajouter une méthode statique sur le type donné, appelée *of*, qui effectuera l'appel à la fonction du framework. Ceci est fait pour fournir un moyen abrégé d'accéder à l'état avec moins de code.

6.2.4.2.2 Gerer l'action « Refuser »

Pour pouvoir gérer l'action de refuser un service, il sera nécessaire de copier le service de la liste des services demandés vers la liste des services refusés. Pour cela dans le fichier « *services.dart* » et plus particulièrement dans la classe *Service*, nous allons ajouter une méthode *copieService*. Cette méthode permettra du dupliquer une instance de service en remplaçant tout ou partie des propriétés de cette instance. Pour cela, nous allons utiliser l'opérateur Null safe « ?? » dans le code ci-dessous :

```

Service copieService(
    String uuid,
    String description,
    DateTime dateEcheance,
    bool accepte,
    DateTime complete,
    Amis amis
) {
    return Service(
        uuid: uuid ?? this.uuid,
        description: description ?? this.description,
        dateEcheance: dateEcheance ?? this.dateEcheance,
        accepte: accepte ?? this.accepte,
        complete: complete ?? this.complete,
        amis: amis ?? this.amis,
    );
}

```

Notez qu'il utilise l'opérateur prenant en charge les valeurs nulles (??) pour créer une nouvelle instance Service avec les valeurs d'origine (si définies), ou celles reçues en tant qu'arguments.

Avec cette méthode, l'action de refuser un service deviendra :

- a) Détruire le service dans la liste des services demandés.
- b) Ajouter une instance duplique du service dans la liste des services refusés.

Pour cela, nous allons créer dans la classe `_PageServiceState` une méthode, nommée `refuserService`, qui recevra l'instance du service refusé :

```
void refuserService(Service service) {
    setState(() {
        servicesDemandes.remove(service);
        servicesRefuses.add(service.copieService(accepte: false));
    });
}
```

Comme vous pouvez le constater, la méthode `setState()` est utilisée ici pour notifier au framework de reconstruire les widgets intéressés. À l'intérieur de son appel, nous supprimons le service de la liste des services demandés et en ajoutons une version modifiée à la liste des services refusés. La version modifiée est obtenue en faisant une copie de la faveur originale et en modifiant sa propriété `accepte`.

Enfin, voici à quoi ressemble le bouton **Refuser** après avoir utilisé la fonctionnalité du paragraphe précédent et la méthode ci-dessus dans la classe `_ElementListeServices` et dans la méthode `_piedDePageElement` :

```
FlatButton(
    child: Text('Refuser'),
    onPressed: () {
        _PageServicesState.of(context).refuserService(service);
    },
),
```

En appelant `_PageServicesState.of(context)`, nous avons accès à l'état actuel du type `PageServiceState` associé au contexte.

6.2.4.2.3 Gérer l'action « Accepter »

Voici à quoi ressemble le bouton **Accepter** après avoir utilisé la technique précédente:

```
FlatButton(
    child: Text('Accepter'),
    onPressed: () {
        _PageServicesState.of(context).accepterService(service);
    },
),
```

Et la méthode `accepterService(service)` correspondante se fait comme suit:

```
void accepterService(Service service) {
    setState(() {
        servicesDemandes.remove(service);
        servicesAcceptes.add(service.copieService(accepte: true));
    });
}
```

Comme vous pouvez le voir, c'est presque la même chose que la méthode `refuserService()`, les seules différences résident dans la liste des cibles et le statut accepté.

6.2.4.3 Appui sur « + » pour demander un service

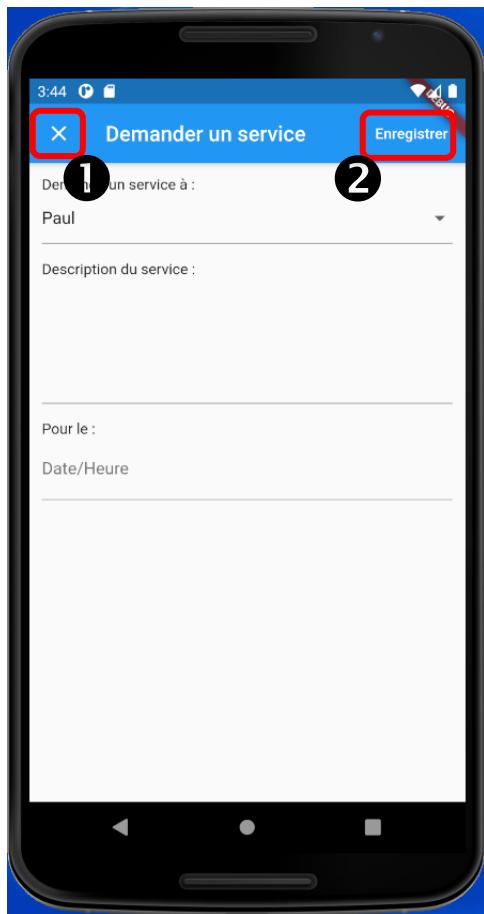
Lorsque l'utilisateur appuie sur le bouton d'action flottant avec le signe plus en bas de la page, il devrait voir le widget `PageDemandeService` à l'écran avec le code que nous avons déjà saisi :

```
floatingActionButton: FloatingActionButton(  
    onPressed: () {  
        Navigator.of(context).push(  
            MaterialPageRoute(  
                builder: (_) => PageDemandeService(  
                    amis: simulAmis,  
                    ),  
                    ),  
                    );  
        },  
        tooltip: 'Demander un service',  
        child: Icon(Icons.add),  
    ),
```

Nous faisons cela en utilisant le widget `Navigator`, qui affiche un nouveau widget à l'écran.

6.2.4.4 Gestion de la page « Demander un service »

Demander un service a quelques gestes à gérer :



Voici comment le processus fonctionne :

- ① Le bouton de fermeture est déjà géré par le widget *CloseButton*, avec le widget *Navigator* (ceci est géré en interne pour nous).
- ② Le bouton **Enregistrer** validera les informations saisies par l'utilisateur et enverra la demande de service à un ami.

6.2.4.4.1 Le bouton de fermeture

Le widget *CloseButton* est intégré à *Navigator*. Il en sort le dernier widget poussé, revenant au précédent. Nous n'avons pas à mettre en œuvre un geste dessus. En utilisant le *Navigator* pour pousser le widget sur l'écran, nous pouvons utiliser le bouton de fermeture pour le supprimer.

6.2.4.4.2 Le bouton « Enregistrer »

Le bouton **Enregistrer** sera chargé de valider et de sauvegarder les nouvelles demandes de service. La sauvegarde sera abordée ultérieurement lorsque nous parlerons de l'intégration Firebase.

Le widget *PageDemandeService* doit également être converti en *StatefulWidget*, car nous aurons besoin de conserver des informations et de manipuler les nouvelles demandes de services avec des actions. Ce sera l'endroit où nous stockerons le service plus tard sur Firebase.

6.2.4.4.3 Validation de la saisie à l'aide du widget *Form*

Nous devons ajouter le widget *Form* à notre mise en page pour pouvoir valider tous les champs à la fois lors de l'enregistrement. Ceci est fait en enveloppant simplement nos widgets de champ de

formulaire avec un widget Form. Nous définissons également la propriété key de notre Form avec une instance `GlobalKey<FormKey>` afin de pouvoir l'utiliser plus tard dans la méthode `enregistrer()`.

```
class PageDemandeService extends StatefulWidget {
  final List<Amis> amis;

  PageDemandeService({Key key, this.amis}) : super(key: key);

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

class _PageDemandeServiceState extends State<PageDemandeService> {
  final _formKey = GlobalKey<FormState>();

  static _PageDemandeServiceState of(BuildContext context) {
    return context.findAncestorStateOfType<_PageDemandeServiceState>();
  }

  @override
  Widget build(BuildContext context) { ... }
}
```

L'appel de la méthode `enregistrer()` ressemble aux précédentes :

```
FlatButton(
  child: Text('Enregistrer'),
  textColor: Colors.white,
  onPressed: () {
    _PageDemandeServiceState.of(context).enregistrer();
  },
),
```

Il recherche dans l'arborescence l'état correspondant et lui demande de sauvegarder. La méthode `enregistrer()` fait le gros du travail:

```
void enregistrer() {
  if (_formKey.currentState.validate()) {
    Navigator.pop(context);
  }
}
```

OK, pour le moment, ça ne fait pas grand-chose. Elle n'appelle que la validation du formulaire correspondant qui parcourt tous les champs du formulaire et les valide, comme vous le savez déjà.

6.3 Thème et style

La création d'interfaces utilisateur avec des thèmes et des styles intégrés donnera à une application un aspect professionnel et facile à utiliser. De plus, le framework permet la création de thèmes et de styles personnalisés et uniques. Pour ce faire, vous apprendrez à personnaliser l'apparence d'une application en ajoutant des polices personnalisées, en utilisant des thèmes et en explorant les célèbres normes de plate-forme, à savoir *iOS Cupertino* et *Google Material Design*. En outre, vous verrez comment utiliser les requêtes multimédias pour un style dynamique.

Chaque application doit avoir sa propre identité. Notre application *service_entre_amis*, par exemple, doit avoir ses propres couleurs et styles. Savoir comment appliquer des styles, des couleurs et des polices personnalisées est fondamental pour y parvenir dans n'importe quelle application. Pour cela, nous allons aborder les sujets suivants :

- ✚ Widgets de thème
- ✚ *Material Design*
- ✚ *iOS Cupertino*
- ✚ Utilisation de polices personnalisées
- ✚ Style dynamique avec *MediaQuery* et *LayoutBuilder*

6.3.1 Les widgets de définition d'un thème

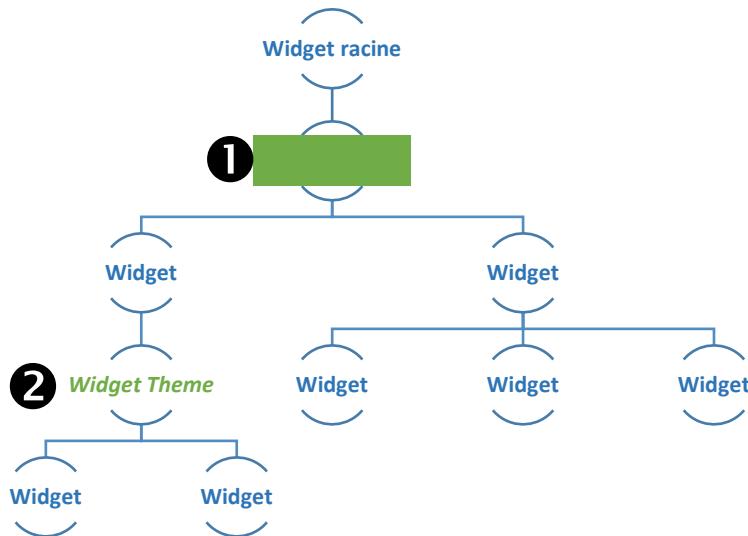
Le développement d'une application va au-delà des bonnes fonctionnalités. Il s'agit également de l'expérience utilisateur que propose l'application. La composabilité des widgets Flutter aide dans cette partie du développement. En définissant une responsabilité unique pour chaque type de widget, nous pouvons choisir de définir des thèmes et des styles qui s'appliquent à un seul widget, à tous les widgets d'un sous-arbre ou à l'ensemble de l'application.

En utilisant le widget *Theme*, nous pouvons personnaliser l'apparence et la convivialité d'une application avec des couleurs personnalisées pour le texte, les messages d'erreur, les surlignages et également les polices personnalisées. Flutter utilise également ce widget dans ses propres widgets.

MaterialApp est un excellent exemple de la manière dont les widgets internes du framework sont composés. Il utilise le widget *Theme* en interne pour personnaliser l'apparence des widgets basés sur *Material Design* tels que les *AppBars* et les boutons. Voyons comment utiliser les widgets *Theme* dans la pratique pour appliquer différents styles à d'autres widgets Flutter.

6.3.1.1 Le widget *Theme*

Dans Flutter, tout est un widget, et nous pouvons construire l'interface utilisateur en ajoutant des widgets en utilisant les propriétés *child* et *children* de chaque widget. Le widget *Theme* se comporte comme n'importe quel autre. Il définit des propriétés et peut avoir un enfant. Le widget *Theme* fonctionne également avec la technique *InheritedWidget*, de sorte que chaque widget descendant peut y accéder en utilisant *Theme.of(context)*, qui appelle en interne la méthode d'assistance *inheritFromWidgetOfExactType* de la classe *BuildContext*. C'est ainsi que les widgets *Material Design* utilisent le widget *Theme* pour définir le style.



Les données du thème sont donc appliquées aux widgets descendants mais peuvent être remplacées dans les parties locales de l'arborescence des widgets. Dans le diagramme précédent, le thème ❷ remplacera le thème ❶ défini au tout début de l'arborescence. Le sous-arbre à partir de ❷ aura un thème différent du reste de l'arbre.

De plus, avec cette structure, il est possible de créer un nouveau thème complet pour certains widgets, ou d'hériter d'un thème de base et de ne modifier que certaines propriétés pour affecter le sous-arbre.

La classe *ThemeData* aide le widget *Theme* dans la tâche de style.

6.3.1.1.1 La propriété *ThemeData*

Le widget *Theme* contient une propriété appelée *data*, qui accepte une valeur *ThemeData* qui contient toutes les informations sur le style, la luminosité du thème, les couleurs, la police, etc.

En utilisant les propriétés de classe *ThemeData*, vous pourrez personnaliser tous les styles liés à l'application, tels que les couleurs, la typographie et des composants spécifiques. Lors de la création de thèmes, vous pouvez choisir de suivre les directives de *Material Design* de Google qui ciblent la conception d'applications pour les appareils mobiles, Web et de bureau, ou *iOS Cupertino* qui sont spécifiques à la plate-forme Apple. Les deux directives de conception ont des singularités dues aux plates-formes cibles.

Le choix de suivre ou non les directives de *Material Design*, *iOS Cupertino* ou aucune d'entre elles vous appartient. Flutter a des widgets basés sur des thèmes conçus pour les deux, afin que vous puissiez appliquer les directives avec précision ou concevoir à votre manière.

La coloration est un sujet important dans la thématisation des widgets. Par exemple, si vous augmentez le contraste du texte sur l'arrière-plan pour mettre en valeur une partie de l'interface utilisateur, il est alors nécessaire d'utiliser les bonnes couleurs. La luminosité est l'une des propriétés clés de la classe *ThemeData* qui vous aidera à manipuler les couleurs.

6.3.1.1.2 La propriété *Brightness*

Une propriété importante d'un thème est la luminosité (brightness). La définition de cette propriété est aussi importante que la définition des couleurs du thème. Comme son nom l'indique, elle expose la luminosité du thème de l'application. Avec cette propriété, les cadres peuvent déterminer le texte, les boutons et les couleurs de surbrillance pour obtenir un contraste suffisant entre le contenu d'arrière-plan et de premier plan.

Elle permet de régler le contraste entre le texte, les boutons et l'arrière-plan (avec les widgets *Material Design*). La classe *ThemeData* a un constructeur *callback()* qui renvoie un thème clair via la valeur *Brightness.light*. Vous pouvez utiliser ses constructeurs *dark()* et *light()* pour essayer vous-même. Lors du choix des couleurs primaires et d'accentuation, il est important d'expérimenter avec les *primaryColorBrightness* et *accentColorBrightness* correspondants. Flutter estime la luminosité en fonction de certains calculs de la luminosité des couleurs, mais il est toujours bon d'expérimenter et de vérifier.

6.3.1.2 Mise en œuvre d'un thème

Le style des widgets dans Flutter peut être effectué de plusieurs manières, et tout ce qui concerne les styles est basé sur un widget *Theme*. Il est temps de voir comment cela fonctionne. Disons, par exemple, que nous avons une application simple, comme suit :

```
class MyAppDefaultTheme extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return Container(  
            color: Colors.red,  
            child: Center(  
                child: Text(  
                    "Simple Text",  
                    textDirection: TextDirection.ltr,  
                ),  
            ),  
        );  
    }  
}
```

Comme vous pouvez le voir, nous utilisons simplement un widget *Container* comme widget racine sans widget *Theme*. Ainsi, nous pouvons supposer que nous n'avons aucun style appliqué à ses widgets descendants. En outre, la propriété *textDirection* est nouvelle à ce stade. Lorsque vous utilisez le widget *MaterialApp* dans notre mise en page, il nous fournit implicitement une valeur *textDirection* par défaut.

Nous pouvons, par exemple, utiliser le widget *Theme* pour changer le style d'un widget *Text*. La classe *ThemeData* contient la propriété *textTheme*, qui contient la configuration du style de texte suivant les instructions de *Material Design* :

```
Text(  
    "Simple Text",  
    textDirection: TextDirection.ltr,  
    style: Theme.of(context).textTheme.headline4,  
,
```

La propriété *style* du widget *Text* accepte une valeur *TextStyle* que nous pouvons obtenir à partir du widget *Theme*. Cependant, comme vous vous en souvenez peut-être, nous n'avons pas spécifié de widget *Theme* dans notre arborescence d'applications. Dans l'exemple précédent, cela fonctionne car la méthode *Theme.of* retourne un widget de secours *ThemeData* quand il n'en est pas défini. Si vous exécutez le code, vous verrez que le widget *Text* est affiché avec une taille de police plus grande que

la taille par défaut. C'est parce que nous utilisons le style *headline4* de *Material Design*. Nous pouvons également personnaliser le style :

```
class MyAppCustomTheme extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Center(
        child: Theme(
          data: Theme.of(context).copyWith(
            textTheme: Theme.of(context).textTheme.copyWith(
              headline4: TextStyle(
                color: Colors.yellow,
              ),
            ),
            child: Text(
              "Simple Text",
              textDirection: TextDirection.ltr,
              style: Theme.of(context).textTheme.headline4,
            ),
          ),
        ),
      );
    );
  }
}
```

Dans ce cas, nous ajoutons un widget *Theme* juste avant le widget *Text* et le personnalisons à l'aide de sa méthode *copyWith*.

- 1) Nous faisons une copie du widget *Theme* par défaut à partir de l'application et ne modifions que sa propriété *textTheme*. La fonction *copyWith* n'est pas obligatoire, cependant, on la voit très souvent lors du développement d'applications Flutter.
- 2) Comme auparavant, cette fois, nous faisons une copie de *textTheme* à partir du thème de base et changeons uniquement sa propriété *headline4* en un nouvel objet de style *Text*. Nous nous attendons à voir du texte jaune, mais ce n'est pas ce que nous voyons, non? C'est parce que nous utilisons le paramètre de contexte à partir du niveau de l'arborescence racine, qui recherchera l'arborescence et ne trouvera pas d'instance de thème, renvoyant la solution de secours, comme nous l'avons vu dans notre premier exemple. Pour que cela fonctionne, nous pouvons utiliser le widget *Builder*, qui déléguera la construction du widget *Text*:

```
class MyAppCustomTheme extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Center(
        child: Theme(
```

```
data: Theme.of(context).copyWith(
    textTheme: Theme.of(context).textTheme.copyWith(
        headline4: TextStyle(
            color: Colors.yellow,
        ),
    ),
),
child: Builder(
    builder: (context) => Text(
        "Simple Text",
        textDirection: TextDirection.ltr,
        style: Theme.of(context).textTheme.headline4,
    ),
),
),
),
),
),
);
}
}
```

Cela fonctionne parce que le widget *Builder* délègue la construction pour qu'elle se produise à un niveau inférieur de l'arborescence, en passant son instance de contexte au niveau inférieur, qui trouvera l'instance de thème correcte lorsqu'il recherchera l'arborescence. Ainsi, lorsque nous exécutons le code précédent, le widget *Text* est affiché avec le style *headline4* correct, qui est presque le même que le style de texte par défaut, seule sa couleur est différente, maintenant jaune.

Comme le thème se réfère au style d'application, nous devons toujours nous soucier de la plate-forme sous-jacente sur laquelle l'application est exécutée. Voyons comment la classe Platform peut vous aider.

6.3.1.3 La classe Platform

Lors du développement d'applications mobiles pour plusieurs plates-formes, nous pouvons avoir besoin de créer des conceptions différentes pour différentes cibles. Pour ce faire, nous pouvons utiliser la classe Platform, qui nous aide à obtenir des informations sur l'environnement, principalement du système d'exploitation cible, via ses propriétés :

- + `isAndroid`
 - + `isFuchsia`
 - + `isIOS`
 - + `isLinux`
 - + `isMacOS`
 - + `isWindows`

Avec ces propriétés, nous pouvons faire en sorte que tout notre arbre de widgets ait des implémentations spécifiques pour chaque plateforme.

```

class PlatformSpecificWidgets extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Platform.isAndroid
      ? MaterialApp(
          theme: ThemeData(primaryColor: Colors.grey),
          home: Material(
            // note the material specific widget
            color: Colors.white,
            child: Center(
              child: Text(
                "Simple Text",
              ),
            ),
          ),
        )
      : CupertinoApp(
          theme: CupertinoThemeData(
            primaryColor: CupertinoColors.lightBackgroundGray,
          ),
          home: Container(
            color: Colors.white,
            child: Center(
              child: Text(
                "Simple Text",
              ),
            ),
          ),
        );
  }
}

```

Comme vous pouvez le voir, en fonction de la plate-forme cible, nous basculons notre widget d'application (et notre thème également) de *MaterialApp* et *ThemeData* (pour Android), à *CupertinoApp* et *CupertinoThemeData* pour toute autre plate-forme cible.

Nous avons vu comment utiliser le widget *Theme* et les classes d'assistance, comme la classe *ThemeData* et *Platform* pour appliquer des styles à nos widgets. Les directives de *Material Design* et *iOS Cupertino* sont présentes dans la base de nombreux widgets de Flutter. Voyons ses fondamentaux pour pouvoir suivre efficacement ces spécifications.

6.3.2 *Material Design*

Material Design est la directive de conception de Google pour aider les développeurs à créer des applications de haute qualité. Elle est présente dans Flutter et évolue toujours avec la plate-forme, avec l'ajout de nouveaux widgets qui suivent les spécifications des composants de *Material Design*. L'importance des styles de *Material Design* pour la plate-forme Flutter est évidente.

Les principaux widgets *Material Design* de Flutter sont *MaterialApp* et *Scaffold*. Les deux aident les développeurs à concevoir une application en suivant les directives de *Material Design* sans trop de travail.

6.3.2.1 Le widget *MaterialApp*

Le premier widget de base pour appliquer les directives *Material Design* dans les applications Flutter est le widget *MaterialApp*. Le widget *MaterialApp* est l'un des deux widgets avec le widget *Theme* qui accepte également une valeur *ThemeData* via sa propriété de *theme*. En plus du thème, *MaterialApp* ajoute des propriétés d'aide pour la localisation, par exemple, ainsi que la navigation entre les écrans.

En ajoutant un widget *MaterialApp* en tant que widget racine de l'application, vous déclarez votre intention de suivre les directives de *Material Design*, et c'est le but, n'est-ce pas ? Maintenant qu'il sait que vous suivez les directives de *Material Design*, le cadre sera légèrement différent par rapport au thème par défaut. Dans le code suivant, nous ne spécifions pas de style pour notre texte :

```
class MaterialAppDefaultTheme extends StatelessWidget {  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            home: Container(  
                color: Colors.white,  
                child: Center(  
                    child: Text(  
                        "Simple Text",  
                        // textDirection: TextDirection.ltr, n'est pas nécessaire  
                        // car défini dans materialapp  
                    ),  
                ),  
            ),  
        );  
    }  
}
```

Dans ce code, nous avons déclaré notre intention de suivre les directives de *Material Design* en ajoutant le widget *MaterialApp* en tant que widget racine. Cela produira une solution de recharge à un style *DefaultTextStyle* peu attrayant pour informer le développeur qu'il n'utilise pas la *Material Design* efficacement dans les widgets de texte (voir affichage ci-contre).

En d'autres termes, nous devrions toujours envelopper les widgets *Text* dans un widget basé sur *Material Design* pour appliquer correctement les styles de typographie proposés par les directives. Le widget *Materiel* est l'exemple le plus simple. Il a une propriété *DefaultTextStyle* et d'autres propriétés de *Material Design* typiques.

Notez que nous n'avons pas non plus fourni de propriété *textDirection* du widget *Text* cette fois. L'une des fonctions de *MaterialApp* est de nous permettre d'appliquer l'internationalisation à notre application, et *textDirection* est basé sur l'environnement local.



En utilisant le widget *MaterialApp*, nous avons vu comment initier les directives de *Material Design* suivantes. Un autre widget important pour vous aider dans cette tâche est le widget *Scaffold*.

6.3.2.2 Le widget *Scaffold*

Nous avons vu précédemment dans le paragraphe « Construction de mises en page dans Flutter », que le widget *Scaffold* a des propriétés qui aident à construire la mise en page avec un look *Material Design*. Son objectif est aussi important que le widget *MaterialApp*. Il aide le développeur à suivre les directives de *Material Design* en ajoutant simplement les widgets correspondants à ses propriétés. L'écran principal de notre application « Service entre amis » suit certains aspects de *Material Design*.



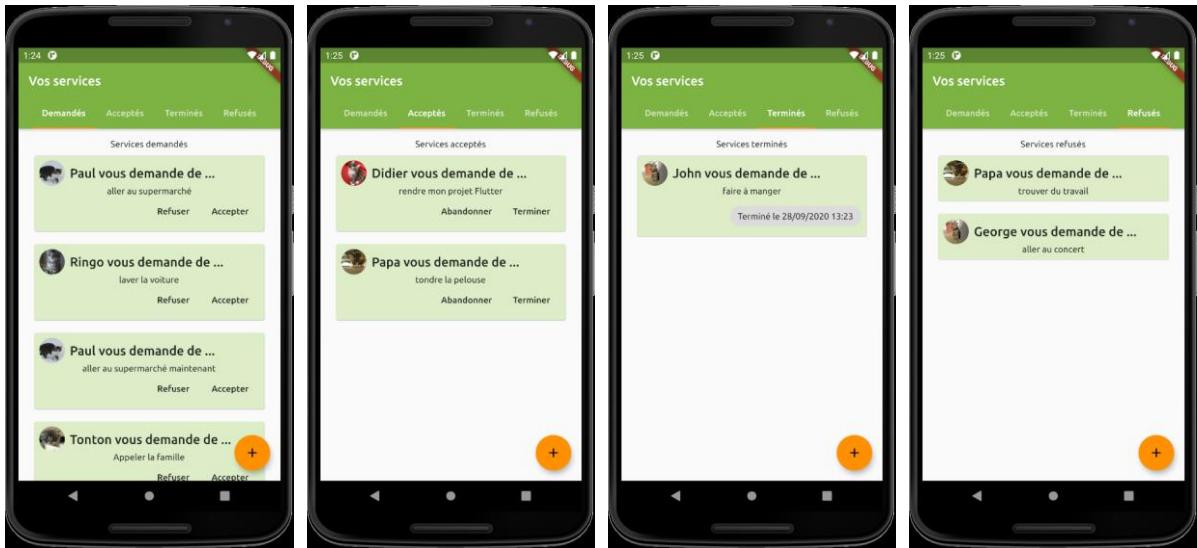
Ici, nous avons utilisé certains composants de *Material Design* ainsi que le widget *Scaffold*. Certains des éléments utilisés sont les suivants :

- ⊕ La barre d'application *AppBar* affichée en haut de l'application contient généralement un titre et des actions de contexte utilisateur, telles que des filtres ou des paramètres. Dans cet exemple, via la propriété *appbar* du widget *Scaffold*, nous montrons un widget *AppBar* qui a un titre et une *TabBar* pour afficher les onglets.
- ⊕ Le bouton d'action flottant *floatingActionButton* est l'un des composants les plus connus de *Material Design*. C'est un bouton rond flottant généralement affiché dans le coin inférieur droit de l'écran. Dans cet exemple, il contient l'action principale de l'application, « Demander un service », en suivant les directives de *Material Design*.

Maintenant que nous avons vu à quoi ressemble le thème par défaut dans certains widgets, voyons comment créer notre propre thème personnalisé avec les couleurs de notre choix.

6.3.2.3 Thème personnalisé

Notre application « Service entre amis » n'a utilisé aucune propriété de *Theme* ou *ThemeData* jusqu'à présent. Il est temps de personnaliser le style de l'application pour la rendre plus attrayante. Voici à quoi il ressemblera après avoir refactorisé ses styles :



Nous allons commencer par créer une définition *themeClair* personnalisée. Il existe plusieurs façons de colorier notre application. Une méthode consiste à définir des couleurs personnalisées pour chacune des propriétés de couleur de la classe *ThemeData* (elle a des propriétés pour chacun des widgets disponibles de *Material Design*). La clé est d'expérimenter les propriétés de couleur et les directives. N'oubliez pas que vous pouvez toujours écraser la définition du thème de l'application dans l'arborescence du widget (avec une couleur de bouton différente, par exemple) en l'enveloppant dans un autre widget de thème.

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

final themeClair = ThemeData(
  primarySwatch: Colors.lightGreen,
  primaryColor: Colors.lightGreen.shade600,
  accentColor: Colors.orangeAccent.shade400,
  primaryColorBrightness: Brightness.dark,
  cardColor: Colors.lightGreen.shade100,
);
```

Nous avons défini un nouveau widget *ThemeData* qui est vert clair par défaut, et nous avons modifié sa propriété *primarySwatch*. Nous avons utilisé une couleur basée sur la palette *Matérial*, où nous pouvons définir certaines couleurs et l'ensemble du schéma sera dérivé de cet échantillon. Bien que le thème par défaut soit clair (fond clair / textes sombres), nous avons défini *primaryColorBrightness* sur *Brightness.dark* afin que le texte qui apparaît au-dessus d'un arrière-plan soit blanc par défaut. Notez également que nous avons défini le thème dans un nouveau fichier Dart (par exemple : *app_theme.dart*) pour faciliter l'organisation du code. Nous devons donc l'importer pour pouvoir l'utiliser dans notre application :

```

import 'app_theme.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: themeClair,
      title: 'Service entre amis',
      home: PageServices(),
    );
  }
}

```

Comme vous pouvez vous y attendre, l'application utilise le *themeClair* importé via sa propriété de *theme*. Pour la définition du schéma de couleurs de l'application, nous avons utilisé l'outil de couleur du site *Web Material Design*.

Changer les couleurs ne suffit pas pour changer l'apparence de l'application. Une autre chose que nous pouvons faire est de changer les styles de texte et d'utiliser les styles *Material Design*. Comme nous l'avons vu précédemment, cela se fait en utilisant la propriété *style* des widgets *Text*. Ainsi, après avoir apporté quelques modifications, nos éléments de liste de services pourraient mettre en valeur certaines parties du texte.

Dans les en-têtes de la liste, par exemple, nous avons ajouté un style pour l'agrandir :

```

Row _enteteElement(BuildContext context, Service service) {
  final styleTitre = Theme.of(context).textTheme.headline6;

```

Nous obtenons le style *styleTitre* du thème de l'application et l'appliquons directement au widget *Text* :

```

child: Text(
  '${service.amis.nom} vous demande de ...',
  style: styleTitre,
),

```

La même chose est faite pour les autres widgets *Text* de l'application. Comme vous pouvez le voir dans notre exemple d'application, il est facile de modifier les styles de notre widget en utilisant le widget *Theme* et les classes d'assistance.

Maintenant que vous connaissez les bases de *Material Design*, introduisons *iOS Cupertino* pour avoir des informations comparatives.

6.3.3 iOS Cupertino

L'objectif de donner à une application un aspect natif est important sur Flutter. Dans cet esprit, beaucoup d'efforts sont faits pour amener le côté *Cupertino* au même niveau de couverture du côté *Material Design*.

L'idée est que le comportement soit fidèle aux applications natives, ce n'est donc pas une tâche facile. La communauté a un rôle important dans cette tâche en utilisant les composants et en donnant des commentaires.

Comme les widgets *Material Design*, *CupertinoApp*, *CupertinoPageScaffold* et *CupertinoTabScaffold* sont les principaux widgets Cupertino disponibles dans Flutter. Nous n'entrons pas dans les détails des widgets *CupertinoPageScaffold* et *CupertinoTabScaffold*. Vous pouvez vérifier ces derniers et tous les widgets Cupertino disponibles en détail <https://flutter.io/docs/development/ui/widgets/cupertino>.

L'alternative *iOS Cupertino* au widget *MaterialApp* est le widget *CupertinoApp*.

6.3.3.1 Le widget *CupertinoApp*

CupertinoApp se comporte de la même manière pour *Cupertino* que *MaterialApp* pour *Material Design*. Il ajoute des fonctionnalités permettant au développeur de suivre les modèles de conception de *Cupertino*. Par exemple, l'application utilise, par défaut, un *bouncing scroll* typique d'iOS, une police personnalisée différente d'Android, etc. En plus du thème, *CupertinoApp* ajoute des propriétés pour la localisation, par exemple, ainsi que la navigation entre les écrans.

Cela fonctionne de la même manière que *Material Design*. Nous pouvons choisir d'utiliser *CupertinoApp* ou non. Ainsi, nous pouvons toujours utiliser les widgets *CupertinoTheme* et *CupertinoThemeData* de la même manière que nous le ferions pour *Material Design*. Ce qui change de l'un à l'autre dans la pratique, ce sont ses propriétés disponibles.

Comme cela est très similaire à la section précédente, nous n'allons pas entrer dans les détails ici. Vous pouvez expérimenter avec les thèmes et jeter un œil au dossier *cupertino_theme* ci-joint pour de petits exemples d'utilisation. Bien que ce ne soit pas recommandé, nous pouvons toujours tout mélanger dans le code, en faisant en sorte que certaines parties suivent *Material Design* et d'autres suivent *Cupertino*. Nous pouvons créer deux classes d'applications, une pour *Material Design* et une pour *Cupertino*. Nous pouvons même créer une classe d'application générique qui modifie la disposition du widget en fonction de la plate-forme (la classe *Platform*).

Expérimetons avec certains des widgets iOS Cupertino dans notre application « Service entre amis ».

6.3.3.2 Utilisation de *Cupertino*

Notre application « Service entre amis » est conçue pour utiliser des composants *Material Design*, mais nous pouvons choisir de la rendre plus native dans *iOS* en utilisant les widgets *Cupertino*. Cela peut être fait avec une combinaison de conditions lors de la construction de nos widgets à l'aide de la classe *Platform*. Nous pouvons, par exemple, concevoir une alternative à *Cupertino* pour le premier écran des services, comme vous pouvez le voir dans la variante *iOS Cupertino*, nous avons la barre de navigation en bas de l'écran. OK, cela n'a pas l'air très bien, mais l'important est de créer des mises en page personnalisées basées sur la plate-forme cible. Flutter vous donne les outils, vous devez les utiliser correctement.



Nous devons vérifier la plate-forme cible et créer différents widgets en fonction de celle-ci. Cela peut être assez compliqué, donc une alternative est de développer des classes de widgets séparées pour chaque plate-forme et de ne pas mélanger tout le code, ce qui facilite l'organisation. Dans notre exemple, nous n'avons fait que le premier écran afin d'illustrer comment l'arbre peut être conditionné en fonction de la plateforme.

```
class MyApp extends StatelessWidget {
    // This widget is the root of your application.
    @override
    Widget build(BuildContext context) {
        if (Platform.isIOS) {
            return CupertinoApp(
                theme: themeClairCupertino,
                home: FavorsPage(),
            );
        }
        return MaterialApp(
            theme: themeClairAndroid,
            home: FavorsPage(),
        );
    }
}
```

L'application « Service entre amis » aura le même style sur les deux plateformes. Voyons maintenant comment utiliser des polices personnalisées pour mettre l'accent sur les applications.

6.3.4 Utilisation de polices personnalisées

Material Design et *Cupertino* fournissent différentes polices pour la conception d'applications, mais il est parfois utile de changer la police par défaut pour une police plus axée sur la marque / le produit.

Comme la police est spécifiée dans le widget *Theme*, nous pouvons l'ajouter au thème de l'application racine, puis elle est appliquée à l'ensemble de l'application. Si vous préférez spécifier la police par widget, cela est également possible. La première étape pour utiliser une police personnalisée dans les applications Flutter consiste à importer des fichiers de polices dans le projet.

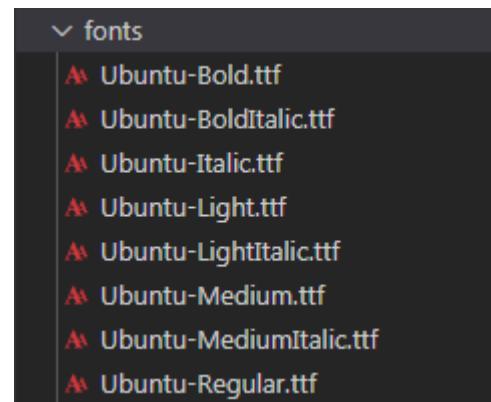
6.3.4.1 Importer des polices dans le projet Flutter

Pour notre premier exemple, nous utiliserons notre application « Service entre amis ». Nous importerons et utiliserons une police personnalisée comme police par défaut pour l'ensemble de l'application.

Pour ce faire, nous pouvons placer les fichiers de polices dans un sous-répertoire de projet et plus tard, déclarer ces fichiers de polices dans le fichier *pubspec.yaml*. Dans cet exemple, nous utiliserons les polices *Ubuntu* trouvées sur le site Web de *Google Fonts* <https://fonts.google.com/>.

La première étape consiste à ajouter les fichiers au répertoire du projet. Il est courant de placer les fichiers de polices dans un sous-répertoire *fonts/* ou *assets/* du projet Flutter. Ici, nous utiliserons un répertoire *fonts/*.

Après cela, nous devons déclarer les polices dans le fichier *pubspec.yaml* afin que le framework sache où trouver la police souhaitée pendant la création du style du texte.



```
flutter:
```

```
# The following line ensures that the Material Icons font is
# included with your application, so that you can use the icons in
# the material Icons class.
uses-material-design: true
fonts:
  - family: Ubuntu
    fonts:
      - asset: fonts/Ubuntu-Regular.ttf
      - asset: fonts/Ubuntu-Italic.ttf
        style: italic
      - asset: fonts/Ubuntu-Medium.ttf
        weight: 500
      - asset: fonts/Ubuntu-Bold.ttf
        weight: 700
```

Comme vous pouvez le voir, nous avons défini la police en quelques sections :

- ❖ **family** : Le champ de famille nomme la police dans le contexte du framework. Il n'est pas nécessaire qu'il corresponde aux noms de fichiers de police. Il sera utilisé pour y faire référence dans le code, alors soyez cohérent.
- ❖ **fonts** : un champ de police suivi d'une liste de champs de la police importée. Toutes les polices spécifiées seront incluses dans l'ensemble de l'application. Nous devons spécifier chaque élément avec les détails correspondant
 - **style** : Ceci détermine si le fichier de ressources correspond à un contour de police normal ou à une variante en italique. Ces valeurs correspondent à l'énumération *FontStyle*.
 - **weight** : Ceci détermine le poids de la police dans l'élément. Il correspond aux valeurs d'énumération *FontWeight* appliquées lors de la mise en page, spécifiez-la donc correctement.

Après avoir importé la police dans le projet, appliquons la police à nos widgets *Text*.

6.3.4.2 Utiliser des polices dans le projet Flutter

Pour utiliser une police sur l'ensemble d'une application, vous devez remplacement la police par défaut dans l'application. L'étape suivante consiste à rendre la police active dans l'application. Nous pouvons le faire dans le thème racine à l'intérieur d'un widget *MaterialApp* et *CupertinoApp*, ou, si nous préférons, nous pouvons ajouter une police directement à un widget *Text* via sa propriété *style* :

```
final themeClair = ThemeData(
  fontFamily: "Ubuntu",
  primarySwatch: Colors.lightGreen,
  primaryColor: Colors.lightGreen.shade600,
  accentColor: Colors.orangeAccent.shade400,
  primaryColorBrightness: Brightness.dark,
  cardColor: Colors.lightGreen.shade100,
);
```

Notre application utilise désormais la famille de polices *Ubuntu* par défaut dans tous les widgets contenant du texte. N'oubliez pas que ce comportement peut être remplacé dans de petites sections de l'application, si vous préférez, en utilisant les widgets *Theme* ou en modifiant directement la propriété de style des widgets *Text*.

Si vous essayez d'utiliser une variante en gras d'une famille de polices personnalisée qui n'est pas déclarée dans le fichier *pubspec.yaml*, le framework utilisera les fichiers génériques pour la police et tentera d'extrapoler les contours pour l'épaisseur et le style demandés.

Comme vous pouvez le voir, vous pouvez appliquer une police personnalisée à l'ensemble de l'application en important simplement la police souhaitée et en la déclarant dans le projet. Un autre aspect important de la thématisation et du style est d'adapter les mises en page pour différents appareils. Les widgets *MediaQuery* et *LayoutBuilder* peuvent vous aider dans cette tâche.

6.3.5 Style dynamique avec MediaQuery et LayoutBuilder

L'adaptation d'une mise en page à une plate-forme peut nous aider à répondre à un public plus large. Mais une autre chose à réaliser est le nombre massif d'appareils différents, ce qui pose d'autres défis aux développeurs.

Développer pour prendre en charge plusieurs tailles d'écran est un défi qui sera toujours présent dans la vie d'un développeur, nous avons donc besoin de mécanismes pour nous adapter de la meilleure façon. Flutter, encore une fois, nous donne les outils dont nous avons besoin pour comprendre

l'écosystème que l'application exécute afin que nous puissions agir dessus. Pour vous aider dans cette tâche, les principales classes fournies par Flutter sont *LayoutBuilder* et *MediaQuery*.

6.3.5.1 Le widget *LayoutBuilder*

Le widget *LayoutBuilder* fournit une propriété de constructeur de type *LayoutWidgetBuilder*. Bien qu'il soit similaire au widget *Builder*, *LayoutWidgetBuilder* contient des informations supplémentaires sur la taille du widget parent dans une valeur *BoxConstraints*.

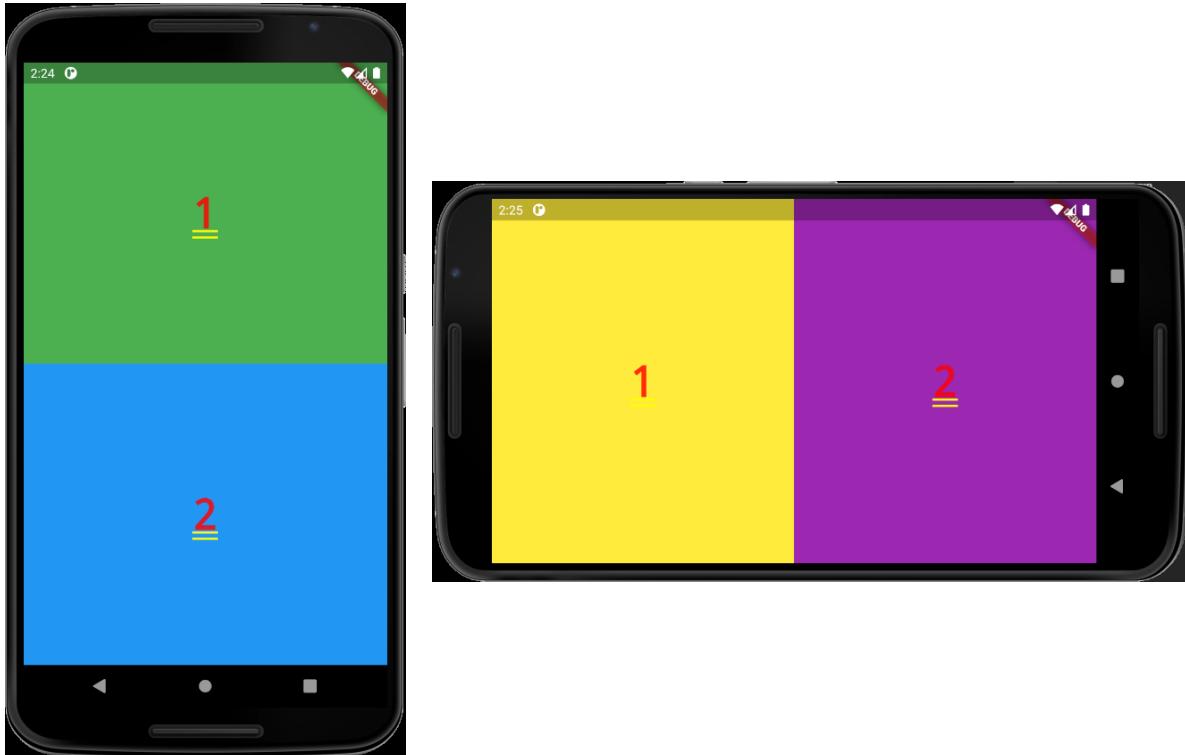
Avec ces informations, la méthode de construction peut être modifiée en fonction de l'espace disponible. Ainsi, dans différents appareils, il y aura une quantité d'espace disponible différente dans le widget racine de l'arborescence, ce qui peut également limiter la taille de ses enfants. En utilisant ce widget, nous pouvons choisir d'afficher ou non certaines parties de la mise en page.

Ce widget dépend de la taille du widget parent, il est donc reconstruit à chaque fois que la taille change. Cela peut se produire de différentes manières sur les appareils mobiles. L'exemple le plus simple est lorsque l'orientation de l'application change, c'est-à-dire lorsque l'utilisateur fait pivoter le téléphone. Voyons comment réagir à un changement de taille à l'écran.

Dans cet exemple, nous allons changer la façon dont deux widgets sont affichés en fonction de l'espace disponible. Ainsi, les widgets sont affichés les uns sur les autres lorsqu'il n'y a pas assez de largeur d'écran (nous évaluons cela à l'aide de l'instance *BoxConstraints* donnée par le widget *LayoutBuilder*), ou côté à côté lorsqu'il y a plus de largeur disponible (comme en position paysage) :

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: LayoutBuilder(
        builder: (BuildContext context, BoxConstraints constraints) {
          if (constraints.maxWidth <= 500) {
            return Column(
              mainAxisAlignment: MainAxisAlignment.max,
              children: <Widget>[
                Expanded(
                  child: Container(
                    color: Colors.green,
                    child: Center(child: Text("1")),
                  ),
                ),
                Expanded(
                  child: Container(
                    color: Colors.blue,
                    child: Center(child: Text("2")),
                  ),
                ),
              ],
            );
          }
          return Row(
            mainAxisAlignment: MainAxisAlignment.max,
            children: <Widget>[
              Expanded(
                child: Container(
                  color: Colors.yellow,
                  child: Center(child: Text("1")),
                ),
              ),
              Expanded(
                child: Container(
                  color: Colors.purple,
                  child: Center(child: Text("2")),
                ),
              ),
            ],
          );
        },
      );
    }
}
```

Comme vous pouvez le voir, nous avons ajouté un widget `LayoutBuilder`, et nous pouvons créer la mise en page en fonction des contraintes (ici si la largeur est inférieure à 500). Conditionnellement, nous affichons un widget `Colonne` lorsque la largeur disponible est inférieure à 500. Et lorsque nous avons assez de place, nous renvoyons un widget `Row`, car nous avons suffisamment d'espace (supérieur à 500). Voici à quoi cela ressemble dans différentes orientations :



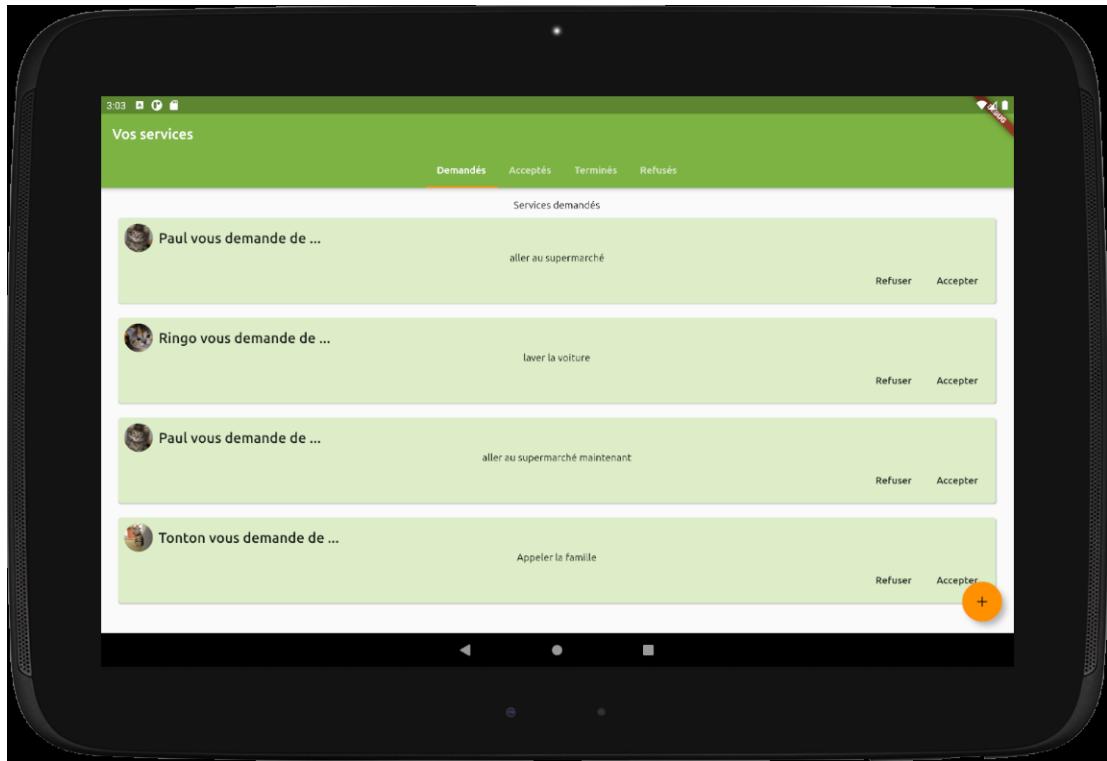
Comme vous pouvez le voir, nous apportons des modifications à la mise en page en fonction non seulement de l'orientation, mais de la largeur disponible. Une autre façon de répondre aux changements de taille disponible consiste à utiliser la classe `MediaQuery`.

6.3.5.2 Le widget `MediaQuery`

`MediaQuery` est un descendant `InheritedWidget` qui contient des informations sur la taille de l'écran et pas seulement le widget parent. En tant que widget `InheritedWidget`, il fournit également la méthode `MediaQuery.of` précédemment introduite, qui recherche dans l'arborescence une instance `MediaQuery`.

Son utilisation est conditionnée par la présence d'une instance dans le contexte. Cela peut être facilement fait en ajoutant une instance de `WidgetsApp` en tant que widget racine. `WidgetsApp` n'est pas spécifique à la plate-forme, comme `MaterialApp` ou `CupertinoApp`, qui utilisent cette classe dans leur implémentation interne. Voyons comment utiliser la classe `MediaQuery` pour créer une mise en page réactive.

Notre application « Service entre amis » n'est pas réactive en termes de taille d'écran jusqu'à présent. Elle affiche une liste verticale d'éléments qui remplit l'espace disponible à l'écran. Pour les smartphones typiques, cela a l'air bien, mais c'est à quoi cela ressemble sur les appareils avec un écran plus grand (par exemple une tablette en format paysage).



Comme vous pouvez le voir, chaque élément remplit chaque ligne et elles sont beaucoup plus grandes que nécessaire. Nous pouvons changer cela en fonction de la taille de l'écran et faire en sorte que la liste affiche plus d'éléments s'il y a plus d'espace que nécessaire pour afficher un service.

En utilisant la classe *MediaQuery*, nous avons effectué un calcul pour modifier le nombre d'éléments affichés par ligne :

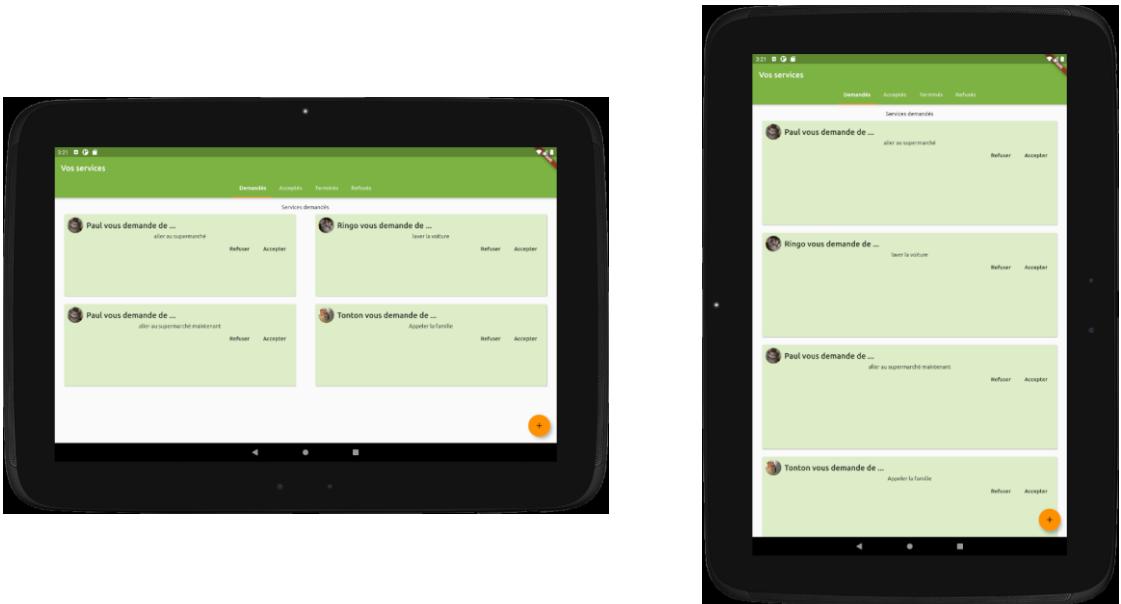
```
import 'dart:math';
const LARGEUR_MAX_ELEMENT = 450.0;
class _ListeServices extends StatelessWidget {
    final String titre;
    final List<Service> services;
    const _ListeServices({Key key, this.titre, this.services})
        : super(key: key);
    @override
    Widget build(BuildContext context) {
        return Column(
            mainAxisSize: MainAxisSize.max,
            children: <Widget>[
                Padding(
                    child: Text(titre),
                    padding: EdgeInsets.only(top: 16.0),
                ),
                Expanded(
                    child: _construireListeElements(context),
                ),
            ],
        );
    }
}
```

```

Widget _construireListeElements(BuildContext context) {
    final largeurEcran = MediaQuery.of(context).size.width;
    final elementsParLigne = max(largeurEcran ~/ LARGEUR_MAX_ELEMENT, 1);
    if (largeurEcran > 400) {
        return GridView.builder(
            physics: BouncingScrollPhysics(),
            itemCount: services.length,
            scrollDirection: Axis.vertical,
            itemBuilder: (BuildContext context, int index) {
                final service = services[index];
                return _ElementListeServices(service: service);
            },
            gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                childAspectRatio: 2.8,
                crossAxisCount: elementsParLigne,
            ),
        );
    } else {
        return ListView.builder(
            physics: BouncingScrollPhysics(),
            itemCount: services.length,
            itemBuilder: (BuildContext context, int index) {
                final service = services[index];
                return _ElementListeServices(service: service);
            },
        );
    }
}

```

Dans le code précédent, en enveloppant la liste des services dans un widget personnalisé pour occuper tout l'espace disponible du widget Column, et nous laissons la logique du redimensionnement avec *MediaQuery* vers la méthode *_construireListeElements()*. Dans cette méthode, vous pouvez voir que nous pouvons diviser la largeur d'écran disponible (*extraite de MediaQuery.of(context).size.width*) par la largeur maximale souhaitée d'un élément (*LARGEUR_MAX_ELEMENT*) et la stocker dans la variable *elementsParLigne*. Plus tard, nous l'utilisons pour vérifier s'il y a de la place pour un autre élément. Ensuite, s'il y a de la place, nous listons les éléments à l'aide d'un widget *GridView* affichant autant de colonnes que *elementsParLigne*. S'il n'y a pas de place pour plus d'une seule carte, nous affichons un widget *ListView* comme auparavant.



Il existe d'autres widgets Flutter disponibles pour cette tâche, alors peut-être qu'une meilleure approche serait d'utiliser un conteneur autre qu'une liste pour afficher les éléments d'une manière plus flexible. D'autres classes peuvent aider sur les ajustements de mise en page.

6.3.5.3 Les classes réactives supplémentaires

Il existe quelques autres widgets qui vous aident à créer des mises en page réactives : *CustomMultiChildLayout* vous donne la liberté de choisir comment un ensemble de widgets enfants est disposé à l'écran à l'aide d'une classe déléguée : *MultiChildLayoutDelegate*. *FittedBox* change sa taille et sa position enfant en fonction d'un ajustement spécifique. *AspectRatio* tente de forcer la taille de son enfant en fonction d'un rapport hauteur / largeur spécifique.

En utilisant toutes ces classes disponibles, nous pouvons rendre nos mises en page Flutter adaptatives. Nous sommes en mesure de personnaliser nos widgets et de personnaliser l'ensemble de l'application.

6.4 Navigation entre les écrans

Les applications mobiles sont généralement organisées sur plusieurs écrans. Dans Flutter, l'itinéraire correspondant à un écran est géré par le widget *Navigator* de l'application. Le widget *Navigator* gère la pile de navigation, en poussant une nouvelle page ou en passant à la précédente.

Dans ce paragraphe, vous apprendrez à utiliser le widget *Navigator* pour gérer les itinéraires de votre application et à ajouter des animations de transition entre les écrans. Les sujets suivants seront traités :

- ✚ Compréhension du widget *Navigator*
- ✚ Compréhension des itinéraires
- ✚ Apprentissage des transitions
- ✚ Exploration des animations Hero

6.4.1 Comprendre le widget *Navigator*

Les applications mobiles contiennent souvent plus d'un écran. Si vous êtes un développeur Android ou iOS, vous connaissez probablement les classes *Activity* ou *ViewController* qui représentent respectivement des écrans sur ces plates-formes.

Une classe importante dans la navigation entre les écrans dans Flutter est le widget *Navigator* qui est responsable de la gestion des changements d'écran avec une idée historique et logique. Un nouvel écran dans Flutter est juste un nouveau widget qui est placé devant un autre. Ceci est géré par le concept de *Routes*, qui définit la navigation possible dans l'application. Comme vous l'avez peut-être déjà deviné, la classe *Route* aide Flutter à travailler sur le flux de navigation.

Les principaux acteurs de la couche de navigation sont les suivants :

- ✚ ***Navigator*** : le widget de gestion de la navigation
- ✚ ***Overlay*** : la navigation l'utilise pour spécifier les apparences des itinéraires
- ✚ ***Route*** : un point final de navigation

6.4.1.1 Introduction

Le widget *Navigator* est le principal acteur dans la tâche de passer d'un écran à un autre. La plupart du temps, nous allons changer d'écran et transmettre des données entre eux, ce qui est une autre tâche importante pour le widget *Navigator*.

La navigation dans Flutter se fait dans une structure de pile. La structure de la pile convient à cette tâche car son concept est très similaire au comportement d'un écran :

- Nous avons un élément en haut de la pile. Dans *Navigator*, l'élément le plus haut de la pile est l'écran actuellement visible dans l'application.
- Le dernier élément inséré est le premier à être retiré de la pile (communément appelé dernier entré, premier sorti (LIFO)). Le dernier écran visible est le premier qui est supprimé.
- Comme un pile, les principales méthodes du widget *Navigator* sont *push()* et *pop()*.

6.4.1.2 Le widget *Overlay*

Dans sa mise en œuvre, *Navigator* utilise le widget *Overlay*. Le widget *Overlay* permet à chacun de ces widgets de gérer leur participation à la superposition à l'aide d'objets *OverlayEntry*.

Nous allons passer par quelques étapes pour vérifier que la manière la plus courante d'utiliser les widgets *Navigator* et *Overlay* est avec les widgets d'application (*WidgetsApp*, *MaterialApp* et *CupertinoApp*) qui offrent plusieurs façons de gérer la navigation dans le widget *Navigator*.

6.4.1.3 La pile de navigation / historique

Comme vous l'avez peut-être déjà remarqué, la méthode `push()` ajoute un nouvel écran en haut de la pile de navigation. `pop()`, à son tour, le supprime de la pile de navigation. Donc, en résumé, la pile de navigation est la pile d'écrans qui sont entrés dans la scène grâce à la méthode `push()` du widget `Navigator`. La pile de navigation est également appelée historique de navigation.

6.4.1.4 Les éléments de la pile : Route

Les éléments de la pile de navigation sont des *Routes*, et il existe plusieurs façons de les définir dans Flutter.

Lorsque nous voulons naviguer vers un nouvel écran, nous y définissons un nouveau widget `Route`, en plus de certains paramètres définis comme une instance `RouteSettings`.

6.4.1.5 La classe `RouteSettings`

Il s'agit d'une classe simple qui contient des informations pertinentes sur la route pour le widget `Navigator`. Les principales propriétés sont les suivantes :

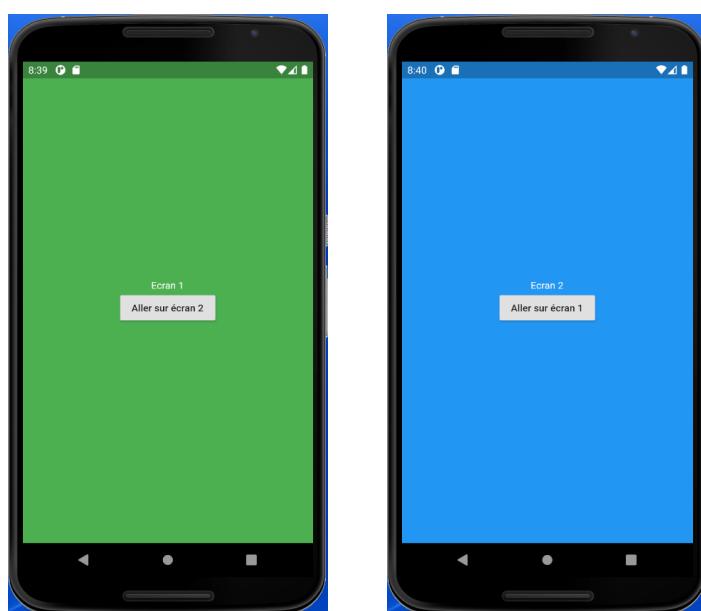
- **`name`** : identifie l'itinéraire de manière unique.
- **`arguments`** : paramètres transmis à l'itinéraire de destination.

6.4.1.6 `MaterialPageRoute` et `CupertinoPageRoute`

La classe `Route` est une abstraction de haut niveau via la fonction de navigation. Cependant, nous ne l'utiliserons pas directement, car nous avons vu qu'un écran est une route dans Flutter et différentes plates-formes peuvent nécessiter de se comporter différemment pour les changements d'écran. Dans Flutter, il existe des implémentations alternatives qui s'adaptent à la gestuelle type de la plate-forme. Ce travail est effectué avec `MaterialPageRoute` et `CupertinoPageRoute`, qui s'adaptent respectivement à Android et iOS. Nous devons donc décider lors du développement d'une application d'utiliser les transitions *Material Design* ou *iOS Cupertino*, ou les deux, en fonction du contexte.

6.4.2 Comment faire pour tout rassembler

Il est temps de découvrir comment utiliser le widget `Navigator` dans la pratique. Créons un flux de base pour accéder à un deuxième écran et inversement (voir exemple ci-dessous) :



6.4.2.1 La navigation directe

La façon de base d'utiliser un widget `Navigator` est comme n'importe quel autre - en l'ajoutant à l'arborescence des widgets :

```
class NavigationDirecte extends StatefulWidget {
    @override
    _NavigationDirecteState createState() => _NavigationDirecteState();
}

class _NavigationDirecteState extends State<NavigationDirecte>
    @override
    Widget build(BuildContext context) {
        return Directionality(
            child: Navigator(
                onGenerateRoute: (RouteSettings settings) {
                    return MaterialPageRoute(
                        builder: (BuildContext context) => _ecran1(context));
                },
            ),
            textDirection: TextDirection.ltr,
        );
    }

    Widget _ecran1(BuildContext context) { ... }

    Widget _ecran2(BuildContext context) { ... }
}
```

Le widget `Navigator` contient une propriété `onGenerateRoute`, un appel qui est responsable de la création d'un widget `Route` basé sur un objet `RouteSettings` passé en argument. Dans l'exemple précédent, vous pouvez voir que nous n'avons pas utilisé l'argument `settings`. A la place, nous avons renvoyé un itinéraire par défaut. L'approche la plus courante vérifierait la propriété `name` des paramètres, qui fonctionne comme l'identificateur de l'itinéraire. Le framework utilise le nom '/' comme itinéraire initial par défaut et fera un appel `initial` en le passant comme argument. Ainsi, l'exemple précédent utilise le widget renvoyé `_ecran1` comme itinéraire initial.

Le résultat de l'appel `onGenerateRoute` est un objet `Route`. Nous avons ici utilisé le type `MaterialPageRoute`. Dans son implémentation la plus basique, nous devrions lui transmettre également un rappel `onGenerateRoute`. Il doit renvoyer un widget à afficher en tant que `Route`. Vous vous demandez peut-être : pourquoi ne pas utiliser une propriété enfant pour ajouter directement le widget enfant ? Sa création dépend du contexte dans lequel il est construit, car le widget `Navigator` peut créer ce widget `Route` dans différents contextes. Mais si vous vérifiez le code suivant, vous verrez que nous pouvons naviguer d'un écran à l'autre en cliquant sur le bouton correspondant. Nous pouvons voir cela dans la méthode `_ecran1` ci-dessous.

```
Widget _ecran1(BuildContext context) {
    return Container(
        color: Colors.green,
        child: Column(
```

```

mainAxisSize: MainAxisSize.max,
mainAxisAlignment: MainAxisAlignment.center,
children: <Widget>[
    Text("Ecran 1"),
    RaisedButton(
        child: Text("Aller sur écran 2"),
        onPressed: () {
            Navigator.of(context).push(
                MaterialPageRoute(
                    builder: (BuildContext context) {
                        return _ecran2(context);
                    },
                ),
            );
        },
    ),
],
),
);
}

```

Ici, vous pouvez vérifier que le widget *Navigator* est accessible en utilisant sa méthode statique *Navigator.of*. Vous serez familier avec cela maintenant et, comme vous le devinez peut-être, c'est ainsi que nous accédons au parent de *Navigator* correspondant à partir d'un contexte spécifique, et oui, nous pouvons avoir de nombreux widgets *Navigator* dans une arborescence. C'est génial, car nous pouvons avoir différents éléments de navigation indépendants dans les sous-sections d'une application.

Revenons à l'exemple, jetons un coup d'œil au rappel *onPressed* du widget *RaisedButton*, où nous poussons une nouvelle *Route* dans la navigation. À partir de là, la valeur que nous transmettons à la méthode *push* est similaire à celle renvoyée par le rappel *onGenerateRoute* dans le navigateur précédemment ajouté.

Pour résumer, notre widget *Navigator* supérieur utilise l'appel *onGenerateRoute* juste pour initialiser la navigation en fournissant l'itinéraire initial. Plus tard, des boutons d'écran ont été ajoutés pour pousser un nouvel itinéraire pour la navigation, en utilisant la méthode *push()* du widget *Navigator*:

```

Widget _ecran2(BuildContext context) {
    return Container(
        color: Colors.blue,
        child: Column(
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                Text("Ecran 2"),
                RaisedButton(
                    child: Text("Aller sur écran 1"),
                    onPressed: () {
                        Navigator.of(context).pop();
                    },
                ),
            ],
        ),
    );
}

```

```

        },
        )
    ],
),
);
}

```

Le widget `_ecran2` est presque égal à `_ecran1`. La seule différence est qu'il sort un itinéraire de la pile de navigation et revient au widget `_ecran1`.

Il y a cependant un problème avec l'exemple précédent. Si nous appuyons sur le bouton de retour sur Android, par exemple, sur l'écran 2, nous devrions revenir à l'écran 1 en conséquence, mais ce n'est pas le cas. Comme nous avons ajouté le widget `Navigator` nous-mêmes, le système n'en a pas conscience. Nous devons également le gérer nous-mêmes.

Pour gérer le bouton de retour, nous devons utiliser `WidgetsBindingObserver`, qui peut être utilisé pour réagir aux messages d'une application Flutter. Nous devons d'abord ajouter `WidgetsBindingObserver` en tant que mixin à notre classe `State`. Nous avons également démarré l'observateur dans `initState()` avec `WidgetsBinding.instance.addObserver(this);` et arrêté l'observateur avec `WidgetsBinding.instance.removeObserver(this);` sur `dispose()`. Avec cette configuration, nous pouvons remplacer la méthode `didPopRoute()` de `WidgetsBindingObserver` et gérer ce qui se passe lorsque le système demande à l'application d'afficher un itinéraire.

La méthode `didPopRoute()` est décrite comme suit dans la documentation: « ***Elle est appelée lorsque le système dit à l'application d'afficher l'itinéraire actuel. Par exemple, sur Android, elle est appelée lorsque l'utilisateur appuie sur le bouton Précédent.*** »

Dans la méthode `didPopRoute()`, nous devons ouvrir un itinéraire à partir de notre widget `Navigator`. Cependant, nous ne pouvons pas accéder à `Navigator` via sa méthode statique, car nous n'avons pas le contexte. Nous devons également ajouter une clé au `Navigator` et accéder à son état (voir code ci-dessous) :

```

class _NavigationDirecteState extends State<NavigationDirecte>
  with WidgetsBindingObserver {
  final _navigatorKey = GlobalKey<NavigatorState>();

  @override
  Future<bool> didPopRoute() {
    return Future.value(_navigatorKey.currentState.canPop());
  }

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
  }
}

```

```

        super.dispose();
    }
...
}

```

Ici, nous avons utilisé la méthode `canPop()` de l'état `Navigator` pour afficher l'itinéraire le plus haut de la navigation.

6.4.2.2 La navigation à l'aide de `WidgetsApp`

Comme nous l'avons vu précédemment, ce n'est pas la manière la plus pratique d'utiliser `Navigator` dans nos applications. Nous avons beaucoup de choses à gérer qui pourraient être évitées.

La façon typique de l'utiliser consiste à utiliser les widgets de l'application. Ils offrent des propriétés et des méthodes pour inclure la navigation dans l'application :

- **`builder`** : nous permet d'ajouter un itinéraire alternatif au navigateur, qui est ajouté par le `WidgetsApp`.
- **`home`** : nous permet de spécifier l'itinéraire vers la page initiale de l'application (normalement '/').
- **`initialRoute`** : nous permet de changer l'itinéraire initial de l'application (par défaut '/').
- **`navigatorKey`** et **`navigatorObserver`** : nous permettent de spécifier les valeurs correspondantes pour le widget `Navigator`.
- **`onGenerateRoute`** : crée des widgets basés sur le nom des paramètres de `Route`, tels que celui utilisé dans l'exemple précédent. C'est l'appel pour créer des `Routes` à partir d'un argument `RouteSettings`.
- **`onUnknownRoute`** : spécifie un appel pour générer un itinéraire en cas d'échec dans un processus de création de `Route` (par exemple, un chemin introuvable).
- **`pageRouteBuilder`** : similaire à `onGenerateRoute`, mais spécialisé sur le type `PageRoute`.
- **`routes`** : Accepte une `Map<String, WidgetBuilder>`, où nous pouvons ajouter une liste de chemin à notre application avec les blocs de construction correspondants.

L'écriture de l'exemple précédent est plus facile car nous pouvons ignorer toutes les implémentations spécifiques au navigateur, comme l'observateur pour le bouton retour ou la touche de navigation :

```

class NavigationWidgetsApp extends StatefulWidget {
    @override
    _NavigationWidgetsAppState createState() => _NavigationWidgetsAppState();
}

class _NavigationWidgetsAppState extends State<NavigationWidgetsApp> {
    @override
    Widget build(BuildContext context) {
        return WidgetsApp(
            color: Colors.blue,
            home: Builder(
                builder: (context) => _ecran1(context),
            ),
            pageRouteBuilder: <Void>(RouteSettings settings, WidgetBuilder builder)
        {
            return MaterialPageRoute(builder: builder, settings: settings);
        }
    }
}

```

```

        },
    );
}

Widget _ecran1(BuildContext context) {
    return Container(
        color: Colors.green,
        child: Column(
            mainAxisAlignment: MainAxisAlignment.max,
            mainAxisSize: MainAxisSize.max,
            children: <Widget>[
                Text("Ecran 1"),
                RaisedButton(
                    child: Text("Aller sur écran2"),
                    onPressed: () {
                        print("CLIC");
                        Navigator.of(context).push(
                            MaterialPageRoute(
                                builder: (BuildContext context) {
                                    return _ecran2(context);
                                },
                            ),
                        );
                    },
                ),
            ],
        ),
    );
}

Widget _ecran2(BuildContext context) {
    return Container(
        color: Colors.blue,
        child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            mainAxisSize: MainAxisSize.max,
            children: <Widget>[
                Text("Ecran 2"),
                RaisedButton(
                    child: Text("Aller sur écran 1"),
                    onPressed: () {
                        Navigator.of(context).pop();
                    },
                ),
            ],
        ),
    );
}

```

Comme vous pouvez le voir, l'implémentation précédente est beaucoup plus simple que la première. Nous spécifions simplement la propriété *home* et *pageRouteBuilder* de l'application, et le reste fonctionne automatiquement :

- Dans *home*, nous définissons l'itinéraire initial de la navigation. Nous l'ajoutons un *builder* pour déléguer sa création à un niveau bas dans l'arborescence, donc quand il recherchera un *navigator*, cela fonctionnera.
- Dans *pageRouteBuilder*, nous définissons le type d'objet *PageRoute* à créer lors de la navigation pour définir l'itinéraire entre les écrans.

6.4.2.3 La navigation à partir d'itinéraires nommés

Le nom de l'itinéraire est un élément important de la navigation. C'est l'identification de l'itinéraire avec son gestionnaire, le widget *Navigator*.

Nous pouvons définir une série d'itinéraires avec des noms associés à chacun d'eux. Ce moyen fournit un niveau d'abstraction à la signification d'un itinéraire et d'un écran. Ils peuvent être utilisés dans une structure de chemin, soit en d'autres termes, ils peuvent être considérés comme des sous-programmes.

6.4.2.3.1 Déplacement vers des itinéraires nommés

Notre exemple précédent utilisant le widget *WidgetsApp* est très simple, mais nous pouvons le transformer en une manière plus organisée de faire les choses. En utilisant des routes nommées, nous pouvons :

- ✚ Organiser les écrans de manière claire
- ✚ Centraliser la création d'écrans
- ✚ Passer des paramètres aux écrans

```
class NavigationItinerairesNommes extends StatefulWidget {  
    @override  
    _NavigationItinerairesNommesState createState() =>  
        _NavigationItinerairesNommesState();  
}  
  
class _NavigationItinerairesNommesState  
    extends State<NavigationItinerairesNommes> {  
    @override  
    Widget build(BuildContext context) {  
        return WidgetsApp(  
            color: Colors.blue,  
            home: Builder(  
                builder: (context) => _ecran1(context),  
            ),  
            routes: {  
                '/2': (context) => _ecran2(context),  
            },  
            pageRouteBuilder: <Void>(RouteSettings settings, WidgetBuilder builder){  
                return MaterialPageRoute(builder: builder, settings: settings);  
            },  
        );  
    }  
}
```

```

Widget _ecran1(BuildContext context) {
    ...
    onPressed: () {
        Navigator.of(context).pushNamed('/2');
    },
    ...
}

Widget _ecran2(BuildContext context) {
    ...
    onPressed: () {
        Navigator.of(context).pop();
    },
    ...
}

```

À partir de l'exemple précédent, vous pouvez voir que nous avons utilisé la propriété *routes* pour définir une table des itinéraires pour que le navigateur sache quoi créer pour chaque itinéraire.

Nous pouvons toujours utiliser la propriété *home* si nous le souhaitons, comme illustré dans l'exemple précédent. Notez que lorsque vous faites cela, nous ne devons pas ajouter la route '/' à la table des itinéraires. Un autre avantage de l'utilisation des itinéraires nommés est de pousser de nouveaux itinéraires. Nous pouvons utiliser la méthode *pushNamed* lorsque nous voulons naviguer vers l'écran 2 depuis l'écran 1. De cette façon, nous n'avons pas besoin de créer l'objet *Route* à chaque appel, nous utiliserons notre constructeur précédemment défini dans la table des itinéraires *routesWidgetsApp*.

6.4.2.3.2 La transmission d'arguments

La méthode *pushNamed* accepte également des arguments, à transmettre au nouvel itinéraire : *Navigator.of(context).pushNamed('/ 2', arguments: "Bonjour de l'écran 1")*; Dans ce cas, nous devons utiliser *onGenerateRoute* de *WidgetsApp* afin d'avoir accès à ces arguments via l'objet *RouteSettings*:

```

class _NavigationItinerairesNommesArgumentsState
    extends State<NavigationItinerairesNommesArguments> {
    @override
    Widget build(BuildContext context) {
        return WidgetsApp(
            color: Colors.blue,
            onGenerateRoute: (settings) {
                if (settings.name == '/') {
                    return MaterialPageRoute(builder: (context) => _ecran1(context));
                } else if (settings.name == '/2') {
                    return MaterialPageRoute(
                        builder: (context) => _ecran2(context, settings.arguments));
                }
            },
        );
    }
}

```

Après cela, nous utilisons l'argument normalement trouvé dans le générateur `_ecran2`, pour afficher un message supplémentaire.

```
Widget _ecran1(BuildContext context) {
    return Container(
        color: Colors.green,
        child: Column(
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                Text("Ecran 1"),
                RaisedButton(
                    child: Text("Aller sur écran 2"),
                    onPressed: () {
                        Navigator.of(context)
                            .pushNamed('/2', arguments: "Bonjour de écran 1");
                    },
                ),
            ],
        ),
    );
}

Widget _ecran2(BuildContext context, String message) {
    return Container(
        color: Colors.blue,
        child: Column(
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                Text("Ecran 2"),
                RaisedButton(
                    child: Text("Aller sur écran 1"),
                    onPressed: () {
                        Navigator.of(context).pop();
                    },
                ),
                Text(message),
            ],
        ),
    );
}
```

6.4.2.4 Récupération des résultats d'un itinéraire

Lorsqu'un itinéraire est poussé dans la pile de navigation, nous pouvons vouloir nous attendre à quelque chose en retour - par exemple, lorsque nous demandons quelque chose à l'utilisateur dans un nouvel itinéraire, nous pouvons prendre la valeur renvoyée via le `pop()` comme résultat de la méthode.

La méthode `pop` et ses variantes renvoient un *Future*. Ce *Future* se résout après l'exécution de la méthode et la valeur de *Future* est le résultat de la méthode `pop()`.

Nous avons vu que nous pouvons passer des arguments à un nouvel itinéraire. Comme le chemin inverse est également possible, au lieu d'envoyer un message au deuxième écran, nous pouvons prendre un message lorsqu'il réapparaît. Dans l'écran 2, nous nous assurons simplement de renvoyer quelque chose lorsque vous faites le *pop* depuis *Navigator* :

```
class NavigationAvecRetour extends StatefulWidget {
  @override
  _NavigationAvecRetourState createState() => _NavigationAvecRetourState();
}

class _NavigationAvecRetourState extends State<NavigationAvecRetour> {
  String _message = "";

  @override
  Widget build(BuildContext context) {
    return WidgetsApp(
      color: Colors.red,
      routes: {
        '/': (context) => _ecran1(context),
        '/2': (context) => _ecran2(context),
      },
      pageRouteBuilder: <Void>(RouteSettings settings, WidgetBuilder builder)
    {
      return MaterialPageRoute(builder: builder, settings: settings);
    },
  );
}

Widget _ecran2(BuildContext context) {
  return Container(
    color: Colors.blue,
    child: Column(
      mainAxisAlignment: MainAxisAlignment.max,
      mainAxisSize: MainAxisSize.max,
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text('Ecran 2'),
        RaisedButton(
          child: Text('Retour sur écran 1'),
          onPressed: () {
            Navigator.of(context).pop("Au revoir de écran 2");
          },
        ),
      ],
    ),
  );
}
```

Le deuxième argument de la méthode *pop* est le résultat de l'itinéraire. Dans l'écran de l'appelant, nous devons reprendre le résultat :

```
Widget _ecran1(BuildContext context) {
    return Container(
        color: Colors.green,
        child: Column(
            mainAxisSize: MainAxisSize.max,
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                Text('Ecran 1'),
                RaisedButton(
                    child: Text('Aller sur écran 2'),
                    onPressed: () async {
                        final message = await Navigator.of(context).pushNamed('/2') ??
                            "Retour via le bouton";
                        setState(() {
                            _message = message;
                        });
                    },
                    ),
                ],
            ),
        );
}
```

Le résultat de *push* est un *Future* que nous devons prendre en utilisant le mot clé *await*. Ici, nous le définissons simplement sur une nouvelle variable *_message* qui s'affiche dans un widget *Text*.

6.4.3 Les transitions d'écran

Le changement d'écran doit être fluide du point de vue de l'expérience utilisateur. Nous avons vu que les widgets *Navigator* fonctionnent sur un widget *Overlay* pour gérer les itinéraires. La transition entre les itinéraires est également gérée à ce niveau.

Comme nous l'avons vu, *MaterialPageRoute* et *CupertinoPageRoute* sont des classes qui ajoutent un itinéraire modale à la superposition avec une transition adaptative à la plate-forme entre l'ancien et le nouvel itinéraire.

Sur Android, par exemple, la transition d'entrée de la page fait glisser la page vers le haut et l'estompe. La transition de sortie fait de même en sens inverse. Sur iOS, la page glisse à partir de la droite et sort en sens inverse. Flutter nous permet également de personnaliser ce comportement en ajoutant nos propres transitions entre les écrans.

6.4.3.1 Le widget *PageRouteBuilder*

PageRouteBuilder est la définition d'une création de *Route*.

Si vous vous en souvenez, `WidgetsApp` contient une propriété `pageRouteBuilder` dans laquelle nous définissons quel `PageRoute` doit être utilisé par notre application et où les transitions sont normalement définies.

`PageRouteBuilder` contient plusieurs callbacks et propriétés pour aider dans la définition de `PageRoute`. Voici quelques exemples :

- **`transitionsBuilder`** : Le callback du constructeur pour la transition, où nous construisons la transition de la route précédente à une nouvelle route.
- **`transitionDuration`** : La durée de la transition
- **`barrierColor`** et **`barrierDismissible`** : Ceci définit les itinéraires partiellement couverts du modèle et non en plein écran

6.4.3.2 La mise en œuvre de transitions personnalisées

Nous pouvons créer une transition personnalisée et l'appliquer globalement dans notre application à l'aide de `pageRouteBuilder` :

```
class _NavigationTransitionState extends State<NavigationTransition> {  
  @override  
  Widget build(BuildContext context) {  
    return WidgetsApp(  
      color: Colors.blue,  
      routes: {  
        '/': (context) => _ecran1(context),  
        '/2': (context) => _ecran2(context),  
      },  
      pageRouteBuilder: <Void>(RouteSettings settings, WidgetBuilder builder)  
    {  
      return PageRouteBuilder(  
        transitionsBuilder:  
          (BuildContext context, animation, secondaryAnimation, widget) {  
            return new SlideTransition(  
              position: new Tween<Offset>(  
                begin: const Offset(-1.0, 0.0),  
                end: Offset.zero,  
              ).animate(animation),  
              child: widget,  
            );  
          },  
        pageBuilder: (BuildContext context, _, __) => builder(context),  
      );  
    },  
  );  
}
```

En faisant cela, nous modifions la transition par défaut d'une classe `MaterialPageRoute` par notre transition personnalisée. Nous procédons comme suit :

- ⊕ Notre `pageRouteBuilder` renvoie maintenant une instance de `PageRouteBuilder`.

- Nous implémentons son callback `pageBuilder` pour renvoyer nos widgets normalement, en appelant le `callback builder`.
- Nous implémentons son appel `transitionBuilder` pour renvoyer un nouveau widget, généralement une instance `AnimatedWidget` ou similaire. Ici, nous retournons un widget `SlideTransition` qui encapsule la logique d'animation pour nous. Une transition de gauche à droite, jusqu'à ce qu'elle devienne pleinement visible.

Une autre façon d'implémenter des transitions personnalisées consiste à créer à la demande des objets `Route`. Dans ce cas, une bonne approche serait d'étendre la classe `PageRouteBuilder` et de créer une transition réutilisable.

6.4.4 Les animations Hero

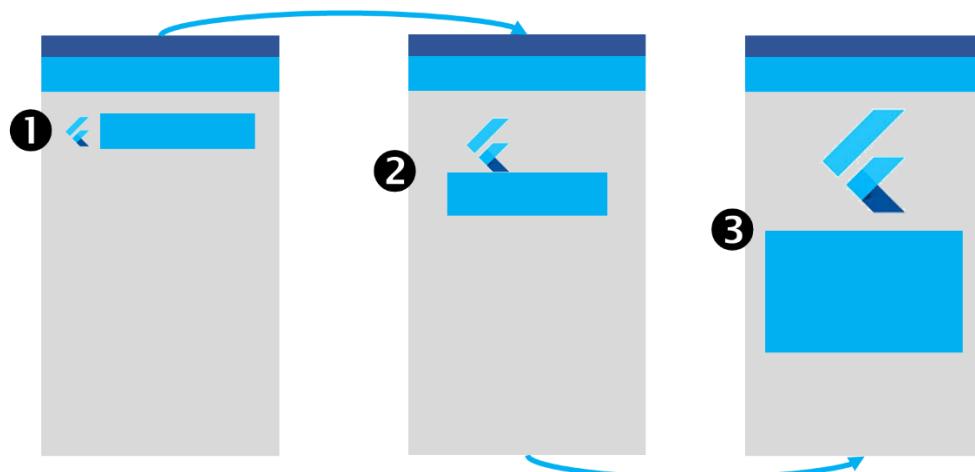
Le nom *Hero* peut paraître étrange au début, mais tous ceux qui ont utilisé une application mobile ont déjà vu ce genre d'animation. Si vous développez pour des plates-formes mobiles, vous avez peut-être déjà entendu parler ou travaillé avec des éléments partagés, c'est-à-dire des éléments qui persistent entre les écrans. C'est la définition d'un *Hero*.

Flutter contient des moyens pour faciliter la création de ce type de mouvement. C'est pourquoi nous pouvons voir comment les animations *Hero* fonctionnent avant même d'approfondir le sujet des animations lui-même.

Le joueur le plus important cette fois est le widget *Hero*. En règle générale, il ne s'agit que d'un seul élément de l'interface utilisateur pour lequel il est logique de voler d'un itinéraire à un autre.

6.4.4.1 Le widget Hero

Dans Flutter, un *Hero* est un widget qui « vole » entre les écrans.

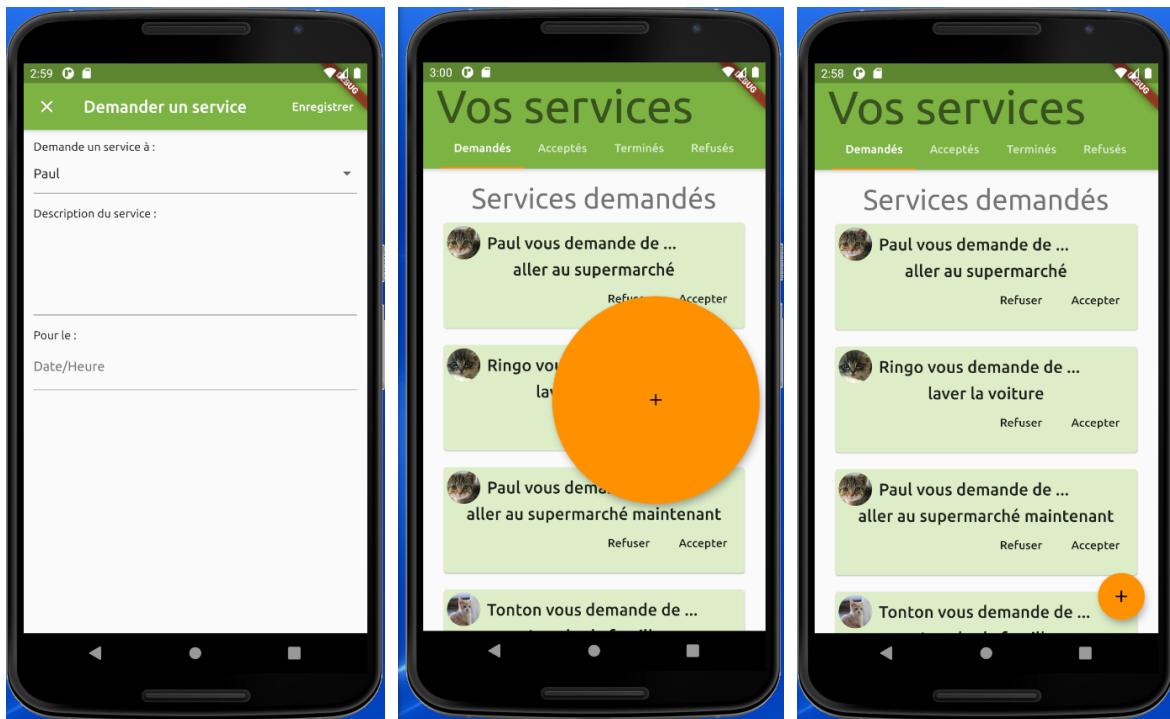


Le widget *Hero*, en réalité, n'est pas le même objet d'écran en écran. Cependant, du point de vue de l'utilisateur, c'est le cas. L'idée est de créer un widget qui vit entre les écrans et change simplement son apparence d'une certaine manière. Comme dans la capture d'écran précédente, l'élément augmente et se déplace en même temps que le nouvel écran apparaît. C'est ce que nous apprenons des trois images de la capture d'écran précédente :

1. Lorsque nous tapons sur un élément de la liste. Par exemple, la transition commence alors que l'écran détaillé est affiché.
2. Une cinématique du processus de transition. Ici, le widget Hero changera sa position et sa taille jusqu'à ce qu'il corresponde au résultat final ③.
3. L'écran final, avec le Hero de l'étape ①, avec une nouvelle taille.

6.4.4.2 Implémentation des transitions avec le widget Hero

Nous allons changer notre application « **Service entre amis** » pour avoir une animation Hero entre l'écran de la liste « **Vos services** » et l'écran « **Demander un service** », de sorte que lorsque nous cliquons sur le bouton flottant « **Demander un service** », il y aura une transition entre celui-ci et l'écran suivant. Le même effet fonctionnera lorsque vous revenez de l'écran « **Demander un service** » à l'écran « **Vos services** ».



Nous commençons le changement en ajoutant un widget Hero à notre arbre. Il devrait envelopper les widgets impliqués dans l'animation.

```
class _PageServicesState extends State<PageServices> {  
    ...  
    @override  
    Widget build(BuildContext context) {  
        ...  
        floatingActionButton: FloatingActionButton(  
            heroTag: "demander_service",  
            onPressed: () {  
                Navigator.of(context).push(  
                    MaterialPageRoute(  
                        builder: (_) => PageDemandeService(  
                            amis: simulAmis,  
                            ),  
                            ),  
                            );  
            },  
            tooltip: 'Demander un service',  
            child: Icon(Icons.add),  
        );  
    }  
}
```

```
        ),
        ),
    );
}
...
}
```

La chose la plus importante à remarquer ici est la simplicité. Notre *FloatingActionButton* contient une propriété *heroTag* qui le fait se comporter comme un widget *Hero*, ce qui signifie qu'il peut animer une transition vers un autre écran.

Pour le deuxième écran, il suffit de répéter le processus :

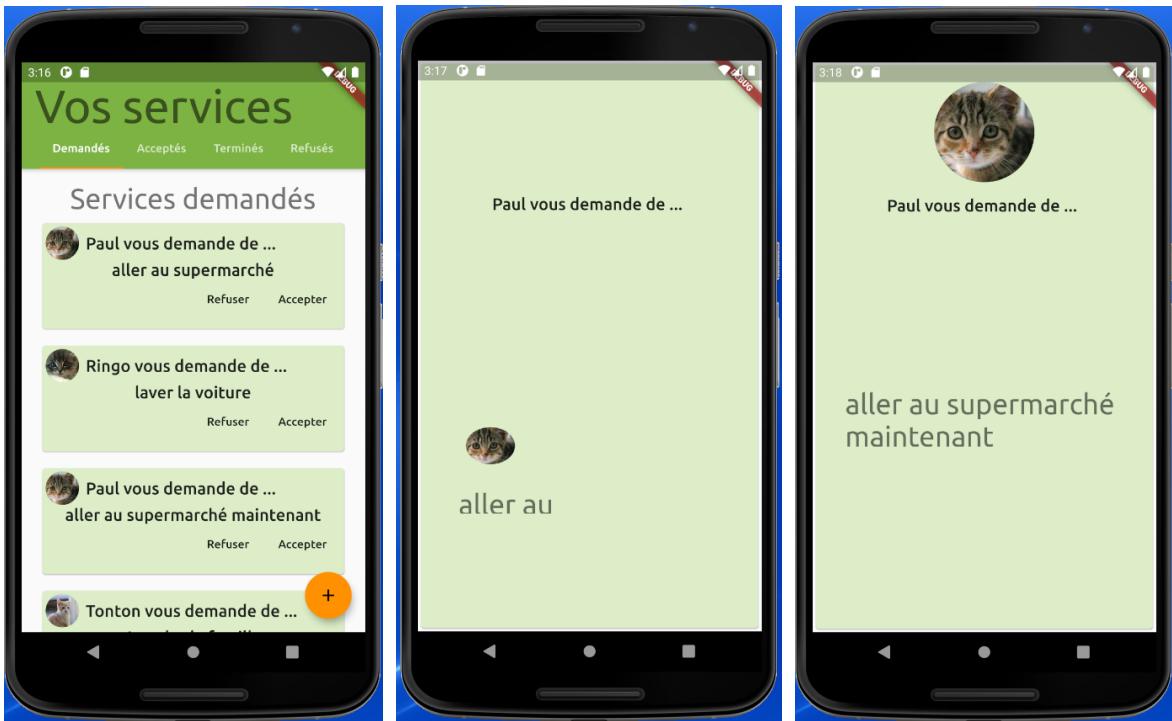
```
class _PageDemandeServiceState extends State<PageDemandeService> {
    ...
    @override
    Widget build(BuildContext context) {
        return Hero(
            tag: "demander_service",
            child: Scaffold(
                ...
            );
        }
        ...
    }
}
```

Faites attention à la propriété *tag*: c'est là que la magie opère.

En outre, il est recommandé que les widgets *Hero* aient des arbres de widgets pratiquement identiques, ou encore mieux, soient le même widget, pour un meilleur résultat d'animation.

Dans notre exemple précédent, nous animions notre *FloatingActionButton* sur l'ensemble du widget d'écran de « **Demander un service** ». Cela fait un effet sympa du bouton au nouvel écran. Cependant, il ne montre pas la meilleure capacité de l'animation *Hero* : partager des éléments entre les écrans. De plus, le widget *FloatingActionButton* et le widget *Scaffold* cible n'ont rien de commun dans sa sous-arborescence de widgets, ce qui fait que notre effet n'est pas le meilleur possible.

Prenons un autre exemple. Supposons que nous ayons un écran de détails pour nos services, et lorsque l'utilisateur appuie sur un service, il affiche la faveur correspondante en plein écran, animant cette transition avec un widget *Hero*. Voici à quoi cela pourrait ressembler :



Je sais que cela n'a peut-être pas l'air cool dans les captures d'écran, mais jetez un œil au code ci-joint pour voir le potentiel du widget *Hero*. Pour que l'avatar et le texte s'animent sur le nouvel écran pendant la transition, nous devons créer deux widgets *Hero*, un pour l'image et un pour la description. Voici ce que nous avons changé dans le widget *_ElementListeServices* :

```
class _ElementListeServices extends StatelessWidget {
  ...
  @override
  Widget build(BuildContext context) {
    final styleDescription = Theme.of(context).textTheme.headline6;
    return Card(
      ...
      children: <Widget>[
        _enteteElement(context, service),
        Hero(
          tag: "description_${service.uuid}",
          child: Text(
            service.description,
            style: styleDescription,
          ),
        ),
        _piedDePageElement(context, service),
      ],
    ),
    padding: EdgeInsets.all(4.0),
  ),
};
```

De la même manière, nous avons modifié la méthode `_enteteElement` pour avoir un widget `Hero` enveloppant notre avatar :

```
Row _enteteElement(BuildContext context, Service service) {
  final styleTitre = Theme.of(context).textTheme.headline6;

  return Row(
    children: <Widget>[
      Hero(
        tag: "avatar_${service.uuid}",
        child: CircleAvatar(
          backgroundImage: NetworkImage(
            service.amis.photoURL,
          ),
        ),
      ),
    ],
  );
}
```

Faites attention à la propriété `tag` de `Hero`. Nous l'avons spécifié en utilisant la valeur `uuid` de service pour rendre le widget `hero` identifiable de manière unique dans le contexte. Pour lancer l'écran des détails des services, nous avons besoin d'un petit changement dans notre widget `_construireListeElements` :

```
Widget _construireListeElements(BuildContext context) {
  final largeurEcran = MediaQuery.of(context).size.width;
  final elementParLigne = max(largeurEcran ~/ LARGEUR_MAX_ELEMENT, 1);
  return ListView.builder(
    physics: BouncingScrollPhysics(),
    itemCount: services.length,
    itemBuilder: (BuildContext context, int index) {
      final service = services[index];
      return InkWell(
        onTap: () {
          Navigator.push(
            context,
            MaterialPageRoute(
              transitionDuration: Duration(seconds: 3),
              pageBuilder: (_, __, ___) =>
                PageDetailsService(service: service),
            ),
          );
        },
        child: _ElementListeServices(service: service),
      );
    },
  );
}
```

Nous avons enveloppé notre `_ElementListeServices` dans un widget `InkWell` pour gérer les appuis dessus. Lorsque l'utilisateur clique dessus, un nouvel itinéraire sera poussé vers le `Navigator` pour afficher le widget `PageDetailsService`.

La dernière partie à examiner est le widget `PageDetailsService`. Ici, nous créons le look final de l'écran des détails d'un service, et en enveloppant l'avatar et la description dans les widgets `Hero`, nous avons une transition impressionnante. Voici à quoi ressemble sa méthode `build()` et, de la même manière, la méthode `_enteteElement()` :

```
class PageDetailsService extends StatefulWidget {
    final Service service;
    const PageDetailsService({Key key, this.service}) : super(key: key);
    @override
    _PageDetailsServiceState createState() => _PageDetailsServiceState();
}

class _PageDetailsServiceState extends State<PageDetailsService> {
    @override
    Widget build(BuildContext context) {
        final styleElement = Theme.of(context).textTheme.headline4;
        return Scaffold(
            body: Card(
                child: Padding(
                    padding: EdgeInsets.symmetric(vertical: 10.0, horizontal: 25.0),
                    child: Padding(
                        child: Column(
                            mainAxisAlignment: MainAxisAlignment.min,
                            crossAxisAlignment: CrossAxisAlignment.stretch,
                            children: <Widget>[
                                _enteteElement(context, widget.service),
                                Container(height: 16.0),
                                Expanded(
                                    child: Center(
                                        child: Hero(
                                            tag: "description_${widget.service.uuid}",
                                            child: Text(
                                                widget.service.description,
                                                style: styleElement,
                                            ),
                                        ),
                                    ),
                                ),
                            ],
                            padding: EdgeInsets.all(10.0),
                        )));
    }
}
```

```

Widget _enteteElement(BuildContext context, Service service) {
  final styleTitre = Theme.of(context).textTheme.headline6;

  return Column(
    mainAxisSize: MainAxisSize.min,
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Hero(
        tag: "avatar_${service.uuid}",
        child: CircleAvatar(
          radius: 60,
          backgroundImage: NetworkImage(
            service.amis.photoURL,
          ),
        ),
      ),
      Container(height: 16.0),
      Text(
        "${service.amis.nom} vous demande de ... ",
        style: styleTitre,
      ),
    ],
  );
}
}

```

Comme vous pouvez le voir, il ressemble au widget `_elementListeService`, visant à avoir des différences minimales dans l'arborescence pour obtenir un meilleur résultat de transition. Notez également que la principale préoccupation est la propriété `tag` de `Hero`, qui doit correspondre à l'origine pour que l'effet fonctionne.

`Navigator` a toujours son importance ici, tout comme les actions `push` ou `pop` qui déclenchent l'animation `Hero` (en signalant que l'itinéraire est en train de changer). Outre la propriété `tag`, `Hero` contient d'autres propriétés pour permettre la personnalisation de l'animation :

- ⊕ **`transitionOnUserGestures`** : Pour activer / désactiver l'animation `Hero` sur les gestes de l'utilisateur tels que le retour sur Android.
- ⊕ **`createRectTween`** et **`flightShuttleBuilder`** : Callbacks pour changer l'apparence de la transition
- ⊕ **`placeholderBuilder`** : Un callback pour retourner un widget qui peut être affiché à la place du widget `Hero` source pendant la transition.

A mesure que nous développons notre compréhension des animations, vous pourrez travailler avec ces propriétés comme un naturel. Les animations de héros sont faciles à implémenter dans Flutter, comme vous pouvez le voir, et même une animation par défaut fournie par le framework peut suffire à créer un effet « agréable » sur certains éléments de mise en page.

7 Accéder à une base de données depuis l'application Flutter

7.1 Introduction

Les développeurs créent généralement des codes modulaires qui peuvent être utilisés dans plusieurs applications. Ce n'est pas différent dans le monde Flutter.

Mes beaucoup d'applications, comme notre application « Service entre amis », doivent pouvoir stocker des informations en local sur l'appareil mobile ou en distant dans le *Cloud* par exemple.

Il existe plusieurs façons d'enregistrer des données sur un appareil mobile. Vous pouvez conserver les données dans un fichier, ou vous pouvez utiliser une base de données locale, telle que SQLite, ou vous pouvez utiliser *SharedPreferences* (sur Android) ou *NSUserDefaults* (sur iOS).

7.2 Accéder à la zone de stockage d'une application (*SharedPreferences*)

Lorsque nous voulons garder peu de données en mémoire avec Flutter, nous pouvons utiliser la bibliothèque *SharedPreferences*. Elle encapsule à la fois *NSUserDefaults* et *SharedPreferences* afin que vous puissiez stocker des données simples de manière transparente à la fois sur iOS et Android sans traiter les spécificités des deux systèmes d'exploitation.

SharedPreferences ne doit pas être utilisé pour les données critiques car les données qui y sont stockées ne sont pas chiffrées et les écritures ne sont pas toujours garanties.

Les données sont toujours conservées sur le disque de manière asynchrone lorsque nous utilisons *SharedPreferences*. *SharedPreferences* est un moyen simple de conserver les données clé-valeur sur le disque.

Par contre, *SharedPreferences* est limitatif car nous ne pouvons stocker que des données primitives, c'est-à-dire des objets variable de base (*int*, *double*, *bool*, *String*) et un objet variable composé (*stringList*). Les données *SharedPreference* sont enregistrées dans l'application, donc, lorsque l'utilisateur désinstalle votre application, les données seront également supprimées.

7.2.1 Implémentation de *SharedPreferences*

Pour utiliser *SharedPreferences*, vous devez inclure *SharedPreferences* dans votre projet. Après une recherche sur internet des informations et version du package, vous devez modifier en conséquence le fichier *pubspec.yaml*.

```
shared_preferences: ^0.5.10
```

Il est ensuite nécessaire d'importer le package dans tous les fichiers qui utiliseront *SharedPreferences*.

```
import 'package:shared_preferences/shared_preferences.dart';
```

7.2.2 Utilisation de *SharedPreferences*

Pour utiliser *SharedPreferences*, vous devez appliquer les points suivants :

- Déclarer une instance de *SharedPreferences* dans votre code

```
SharedPreferences preferences;
```

- Pour écrire des données dans *SharedPreferences*, vous devez dans un premier temps initialiser l'instance de *SharedPreferences* en utilisant la méthode *getInstance()*. Cette méthode est asynchrone (*async*) et devra être utilisée dans une fonction asynchrone qui renverra un objet *Future*. Pour suspendre l'exécution jusqu'à la fin d'un *Future*, vous utiliserez *await* dans une

fonction `async`. `SharedPreferences.getInstance()` est asynchrone, vous devez donc utiliser l'instruction `await` pour nous assurer que l'instance a été initialisé avant l'exécution des prochaines lignes de code.

Pour écrire la valeur dans le stockage persistant (`SharedPreferences`), vous devez utiliser l'une des méthodes ci-dessous en fonction du type de la variable contenant la donnée à sauvegarder :

- `setBool` (clé de chaîne, valeur booléenne) → Future <bool>
 - Enregistre une valeur booléenne en arrière-plan.
- `setDouble` (clé de chaîne, valeur double) → Future <bool>
 - Enregistre une valeur double en arrière-plan.
- `setInt` (clé de chaîne, valeur int) → Future <bool>
 - Enregistre une valeur entière en arrière-plan.
- `setString` (clé de chaîne, valeur de chaîne) → Future <bool>
 - Enregistre une valeur de chaîne en arrière-plan.
- `setStringList` (clé de chaîne, valeur de <String> de liste) → Future <bool>
 - Enregistre une liste de valeurs de chaîne en arrière-plan. [...]

```
ecrireParametres(int param ) async {
    preferences = await SharedPreferences.getInstance();
    if (param != null) {
        await preferences.setInt('PARAM', param);
    }
}
```

c) Pour lire des données dans `SharedPreferences`, vous devez dans un premier temps initialiser l'instance de `SharedPreferences` en utilisant la méthode `getInstance()` comme précédemment. Pour lire la valeur dans le stockage persistant (`SharedPreferences`), vous devez utiliser l'une des méthodes ci-dessous en fonction du type de la variable contenant la donnée à lire :

- `get` (clé de chaîne) → dynamique
 - Lit une valeur de n'importe quel type.
- `getBool` (clé de chaîne) → booléen
 - Lit une valeur de type booléen.
- `getDouble` (clé de chaîne) → double
 - Lit une valeur de type double.
- `getInt` (clé de chaîne) → int
 - Lit une valeur de type int.
- `getString` (clé de chaîne) → chaîne
 - Lit une valeur de type chaîne de caractères.
- `getStringList` (clé de chaîne) → Liste <String>
 - Lit un ensemble de valeurs de chaîne.

```
int lireParametres() async {
    preferences = await SharedPreferences.getInstance();
    int param = preferences.getInt('PARAM');
    return param ;
}
```

D'autres méthodes de l'instance SharedPreferences peuvent être utilisées comme :

- `getKeys () → Liste <String>`
 - Renvoie toutes les clés de l'instance utilisée.

7.3 Accéder à une base de données interne SQLite

Autre solution locale de stockage est d'utiliser SQLite pour stocker des données dans une base de données locale.

L'exemple que nous allons utiliser dans ce paragraphe sera une simple application de gestion d'une liste de course. Application utile si vous êtes comme moi qui oubliais des choses de temps en temps, elle pourrait vous aider à revenir du supermarché avec toutes les courses dont vous avez besoin.

Nous allons construire une application de base de données entièrement fonctionnelle ou vous verrez :

- ⊕ Utiliser SQLite dans Flutter.
- ⊕ Créer des classes modèles.
- ⊕ Afficher les données aux utilisateurs de l'application.
- ⊕ Utiliser des singletons et effectuer des actions de création (**Create**), de lecture (**Read**), de mise à jour (**Update**) et de suppression (**Delete**) sur une base de données locale autrement appelé les actions **CRUD**.

7.3.1 Rappel sur la théorie des Bases de Données Relationnelles (BDR)

Selon le site officiel (SQLite.org), SQLite est un «*petit moteur de base de données SQL rapide, autonome, hautement fiable et complet*». Voyons ce que cela signifie pour nous, en tant que développeurs mobiles Flutter.

Premièrement, SQLite est un moteur de base de données SQL. Cela signifie que vous pouvez utiliser le langage SQL pour créer des requêtes, donc si vous êtes déjà familiarisé avec SQL, vous pouvez tirer parti de vos connaissances. Si vous êtes totalement nouveau dans les bases de données, je vous suggère de jeter un coup d'œil aux nombreux tutoriels sur le langage SQL présent sur internet. Vous trouverez beaucoup plus facile le suivi de ce chapitre.

Deuxièmement, les principales caractéristiques de SQLite sont les suivantes :

Petit et rapide : les développeurs ont largement testé la vitesse et la taille des fichiers de SQLite, et il a surpassé plusieurs autres technologies, à la fois en termes d'espace sur le disque et pour sa vitesse de récupération des données.

Autonome signifie que SQLite nécessite très peu de bibliothèques externes, ce qui en fait le choix idéal pour toute application légère et indépendante de la plate-forme. SQLite lit et écrit directement à partir des fichiers de base de données sur le disque, vous n'avez donc pas à configurer de connexion client-serveur pour l'utiliser.

Haute fiabilité : SQLite a été utilisé sans problème dans plusieurs milliards d'appareils mobiles, Internet des objets (IoT) et de bureau depuis plus d'une décennie, se révélant extrêmement fiable.

Fonctionnalités complètes : SQLite dispose d'une implémentation SQL complète, comprenant des tables, des vues, des index, des déclencheurs et plusieurs fonctions SQL standard.

SQLite est un très bon choix pour les données persistantes sous Android et iOS car elle est facile à implémenter et sécurisée dans le domaine public, multiplateforme et compact.

Afin d'ajouter les fonctionnalités SQLite dans Flutter, nous utiliserons le plugin *sqflite*, qui est le plugin SQLite pour Flutter qui prend actuellement en charge iOS et Android, et contient des méthodes d'assistance asynchrones pour les requêtes *SELECT*, *INSERT*, *UPDATE* et *DELETE*. Nous verrons les étapes requises pour utiliser la bibliothèque de plugins *sqflite* tout au long de ce chapitre. La base de données que nous allons créer comporte deux tables : une table des listes de courses et une table des éléments de listes de courses.

La table des listes de courses (nommée *courses*) comporte 3 champs :

- 1) id (entier)
- 2) nom (texte)
- 3) priorite (entier).

La table des éléments (nommée *articles*) comporte 5 champs :

- 1) id (entier)
- 2) nom (texte)
- 3) quantite (texte)
- 4) commentaire (texte)
- 5) idCourse qui est un id sur la liste de courses (entier).

Ce dernier est la clé étrangère qui pointe vers l'id de la liste de courses. Comme vous pouvez le voir, le schéma est très simple, mais il nous permettra d'expérimenter de nombreuses fonctionnalités nécessaires pour créer une application de base de données.

7.3.2 Créer une base de données SQLite

Dans un premier temps, créons un nouveau projet Flutter depuis votre éditeur. Nous pouvons appeler « *listes_courses* ».

La première étape est, dans notre projet, d'ajouter la dépendance du package *sqflite* dans le fichier *pubspec.yaml*. Comme d'habitude, afin de trouver la dernière version de la dépendance, veuillez visiter <https://pub.dev/packages/sqflite>. Nous aurons besoin d'un deuxième package qui est *path*. Ce dernier nous permet de trouver le chemin par défaut pour stocker notre base de données sur notre appareil.

En effet, *path.dart* est une bibliothèque qui vous permet de manipuler les chemins de fichiers. Ceci est utile ici, car chaque plateforme (iOS ou Android) enregistre le fichier dans des chemins différents. En utilisant la bibliothèque *path.dart*, nous n'avons pas besoin de savoir comment les fichiers sont enregistrés dans le système d'exploitation actuel, et nous pouvons toujours accéder à la base de données en utilisant le même code.

```
dependencies:  
  flutter:  
    sdk: flutter  
  sqflite: ^1.3.1  
  path: ^1.7.0
```

Afin d'apprendre à mieux structurer notre projet, nous allons créer dans le dossier *lib*, un sous-dossier appelé *util*. Dans ce dossier, créez un nouveau fichier : *gestion_bdd.dart*. Ce fichier contiendra les

méthodes pour créer la base de données et pour lire et écrire des données. En haut de ce fichier, nous importerons *sqflite.dart* et *path.dart*.

```
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
```

A partir de là, nous allons créer une nouvelle classe qui peut être appelée à partir d'autres parties de notre code. De manière tout à fait prévisible, nous pouvons l'appeler *GestionBdD*. En haut de cette classe, créez deux variables :

- ⊕ *version* : qui est de type entier, représentant la version de la base de données, qui au début est 1. Cela facilitera la mise à jour de la base de données lorsque vous devez modifier quelque chose dans sa structure.
- ⊕ *bdd* : qui est de type *Database*, contiendra la base de données SQLite elle-même.

```
class GestionBdD {
    final int version = 1;
    Database bdd;
    ...
}
```

Dans cette classe, nous allons ajouter des méthodes permettant de créer, tester, lire et écrire.

La première méthode est *ouvrirBdD* qui ouvrira la base de données si elle existe ou la créera si ce n'est pas le cas. Les opérations de base de données pouvant prendre un certain temps à s'exécuter, en particulier lorsqu'elles impliquent de traiter une grande quantité de données, elles sont asynchrones. Par conséquent, la fonction *ouvrirBdD* sera asynchrone et renverra une base de données de type *Future*.

Pour continuer à structurer notre programme, nous allons définir les requêtes dans des constantes de type *String*. Les deux premières requêtes font appel à l'instruction SQL « *CREATE TABLE* » : la première pour créer la table des listes de courses, et la deuxième pour la table des articles de ces listes.

```
const String reqCreerTableListes =
    'CREATE TABLE courses (id INTEGER PRIMARY KEY, nom TEXT, priorite INTEGER)
';
const String reqCreerTableArticles =
    'CREATE TABLE articles (id INTEGER PRIMARY KEY, idCourse INTEGER, nom TEXT
, quantite TEXT, commentaire TEXT, FOREIGN KEY(idCourse) REFERENCES courses(id
))';
```

Vous remarquerez peut-être que nous n'utilisons que deux types de données : *INTEGER* et *TEXT*. Dans SQLite, il existe seulement cinq types de données : *NULL*, *INTEGER*, *REAL*, *TEXT* et *BLOB*. Notez qu'il n'y a pas de types de données Boolean ou Date. Le champ de quantité de la table des articles est un *TEXT* et non un nombre car nous voulons permettre à l'utilisateur d'insérer également la mesure, par exemple "5 pièces" ou "2 kg". Lorsqu'un champ entier est appelé *id* et est une clé primaire, lorsque vous fournissez *NULL* lors de l'insertion d'un nouvel enregistrement, la base de données attribuera automatiquement une nouvelle valeur, avec une logique d'auto-incrémantation. Donc, si la plus grande valeur de l'*id* est 10, l'enregistrement suivant prendra automatiquement 11.

Dans la méthode ***ouvrirBdD()***, nous devons d'abord vérifier si l'objet *bdd* est nul. C'est parce que nous voulons éviter d'ouvrir une nouvelle instance de la base de données inutilement. Dans la méthode ***ouvrirBdD()***, ajoutons le code suivant :

```
Future<Database> ouvrirBdD() async {
    if (bdd == null) {
        bdd = await openDatabase(join(await getDatabasesPath(), 'courses.db'),
            onCreate: (database, version) {
                database.execute(reqCreerTableListes);
                database.execute(reqCreerTableArticles);
            }, version: version);
    }
    return bdd;
}
```

Si *bdd* est nul, nous devons ouvrir la base de données. La bibliothèque *sqflite* a une méthode *openDatabase*. Nous définirons trois paramètres dans notre appel :

- ⊕ le chemin de la base de données à ouvrir,
- ⊕ la version de la base de données
- ⊕ le paramètre *onCreate*.

Le paramètre *onCreate* ne sera appelé que si la base de données au chemin spécifié est introuvable ou si la version est différente. La fonction à l'intérieur du paramètre *onCreate* prend deux valeurs : une base de données et une version.

Dans la fonction, nous appelons deux fois la méthode *execute()* qui exécute des requêtes SQL brutes dans une base de données. Enfin, retournons la base de données à la fin de la méthode ***ouvrirBdD()***. Nous allons ensuite vérifier que cela fonctionne comme prévu.

Pour le moment, même si nous appelions la méthode ***ouvrirBdD()***, nous n'aurions aucun moyen de savoir si la base de données a été correctement créée ou non. Afin de tester la base de données, nous allons créer une méthode dans la classe *GestionBdD* qui insérera un enregistrement dans la table des courses, un enregistrement dans la table des articles, puis récupérera les deux et les imprimera dans la console de débogage. Enfin, nous allons modifier la méthode principale pour qu'elle appelle la méthode de test. De cette façon, nous nous assurerons que la base de données a été créée correctement et que nous pouvons y lire et écrire des données.

Pour cela, nous devons ajouter une méthode *testBdD()* dans la classe *GestionBdD*. Cette méthode insérera des données simulées dans notre base de données, puis récupérera les données et les imprimera dans la console de débogage. Toutes les méthodes de base de données sont asynchrones, donc *testBdD ()* renvoie un Future et est marqué comme asynchrone.

Pour continuer à structurer notre code nous allons créer deux nouvelles constantes String pour deux nouvelles requêtes utilisant l'instruction SQL « *INSERT* ».

```
const String reqCreerListe =
    "INSERT INTO courses (id, nom, priorite) VALUES (1, 'Epicerie', 2)";
const String reqCreerArticle =
    "INSERT INTO articles (id, idCourse, nom, quantite, commentaire) VALUES (1
, 1, 'Pommes', '2 Kg', 'Meilleures vertes');
```

La méthode `testBdD()` a le code suivant :

```
Future testBdD() async {
  bdd = await ouvrirBdD();
  await bdd.execute(reqCreerListe);
  await bdd.execute(reqCreerArticle);
  List lists = await bdd.rawQuery(reqLireListes);
  List items = await bdd.rawQuery(reqLireArticles);
  print(lists[0].toString());
  print(items[0].toString());
}
```

Tout d'abord, nous attendons la méthode `ouvrirBdD()`, qui renvoie la base de données. La première fois que vous appelez cette méthode, la base de données est créée. Ensuite, nous appelons la méthode `execute`. Nous l'appelons la première fois pour insérer un enregistrement dans la table des courses, et la deuxième fois, nous l'appelons pour insérer un enregistrement dans la table des articles. Dans les deux cas, nous utilisons le langage SQL, avec une requête d'insertion.

Ensuite, nous lisons les tables de la base de données à l'aide de la méthode `rawQuery` et en passant une requête de lecture utilisant l'instruction SQL « `SELECT` » que nous définissons dans deux nouvelles constantes.

```
const String reqLireListes = 'SELECT * FROM courses';
const String reqLireArticles = 'SELECT * FROM articles';
```

« `SELECT *` » prend toutes les valeurs de la table spécifiée. Nous renvoyons les valeurs récupérées dans une liste. Enfin, nous imprimons dans la console de débogage le premier élément des deux listes que nous avons remplies avec la méthode `rawQuery`.

Il ne reste plus qu'à appeler cette méthode dans notre page principale de notre application en oubliant pas d'importer notre nouvelle classe en haut du fichier.

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Exemple utilisation SQLite',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
```

```

        home: MyHomePage(title: 'Listes de courses'),
    );
}
}

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) {
        GestionBdD gestionBdD = GestionBdD();
        gestionBdD.testBdD();

        return Scaffold(
            appBar: AppBar(
                title: Text(widget.title),
            ),
        );
    }
}

```

Nous sommes maintenant prêts à essayer l'application : si vous l'exécutez et regardez la console de débogage, vous devriez voir le résultat de la requête SELECT que nous avons effectuée sur la base de données.

```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Launching lib\main.dart on sdk gphone x86 64 arm64 in debug mode...
↳ Built build\app\outputs\flutter-apk\app-debug.apk.
Connecting to VM Service at ws://127.0.0.1:59762/e8xvP5oiQ_M=/ws
I/flutter (26790): {id: 1, nom: Epicerie, priorite: 2}
I/flutter (26790): {id: 1, idCourse: 1, nom: Pommes, quantite: 2 Kg, commentaire: Meilleures vertes}

```

Cela signifie que nous avons correctement inséré et récupéré les données des deux tables (courses et articles) dans notre base de données. C'est un bon point de départ, mais pour le moment, notre application n'affiche qu'un écran blanc.

Veuillez noter que si vous exécutez l'application plusieurs fois, vous obtiendrez une exception SQL en raison d'un échec de contrainte, car vous insérerez des enregistrements avec la même clé de ligne unique dans les tables. Modifiez simplement la valeur de l'ID dans l'instruction d'insertion pour ajouter plus de données.

De plus, si vous exécutez plusieurs fois l'application en ayant modifié la structure de la base, il est nécessaire dans ce cas de déinstaller l'application de votre émulateur ou de votre mobile pour supprimer l'ancienne base de données.

Dans le paragraphe suivant, nous commencerons le processus en créant les classes de modèle que nous utiliserons dans notre code pour interagir plus efficacement avec la base de données.

7.3.3 Création des classes de modèle pour accéder à une base de données

Une approche courante lorsqu'on traite une base de données à partir d'un langage de programmation orienté objet (ou POO pour faire court) consiste à traiter des objets qui reflètent la structure des tables d'une base de données. Cela rend le code plus fiable, plus facile à lire et aide à éviter les incohérences dans les données.

Notre structure *courses.db* est extrêmement simple, nous devrons donc simplement créer deux classes de modèle, contenant les mêmes champs que ceux qui sont maintenant dans les tables, et une méthode de Map pour simplifier le processus d'insertion et d'édition des données dans la base de données.

Pour cela, nous allons créer un sous-répertoire *modeles* dans *lib*. Dans ce dossier *modeles*, créez un nouveau fichier appelé *liste_courses.dart*. Dans ce fichier, créez une classe appelée **Course** qui contiendra trois propriétés ainsi que le constructeur associé

- ✚ Un entier id,
- ✚ Une String nom
- ✚ Un entier priorite,

Enfin, nous allons créer une méthode **toMap()** qui retournera une *Map* de type *String, dynamic*. Pour rappel, une *Map* est une collection de paires clé/valeur : le premier type que nous spécifions est pour la clé, qui dans ce cas sera toujours une chaîne. Le second type concerne la valeur : comme nous avons différents types dans le tableau, celle-ci sera dynamique. Dans une *Map*, vous pouvez récupérer une valeur à l'aide de sa clé.

```
class ListeCourses {  
    int id;  
    String nom;  
    int priorite;  
    ListeCourses(this.id, this.nom, this.priorite);  
  
    Map<String, dynamic> toMap() {  
        return {  
            'id': (id == 0) ? null : id,  
            'nom': nom,  
            'priorite': priorite,  
        };  
    }  
}
```

Dans une base de données *SQLite*, lorsque vous fournissez une valeur nulle lorsque vous insérez un nouvel enregistrement, la base de données attribuera automatiquement une nouvelle valeur, avec une logique d'incrémentation automatique. C'est pourquoi pour l'id, nous utilisons un opérateur ternaire : lorsque l'id est égal à 0, nous le changeons en null, afin que *SQLite* puisse définir l'id pour nous. Pour les deux autres champs, la *Map* prendra les valeurs de la classe. Maintenant, nous pouvons répéter les mêmes étapes pour la classe de modèle Article.

Pour cela, vous créez un nouveau fichier appelé *article.dart* dans le dossier *modeles*. Dans le fichier, créez une classe appelée ***Article*** qui contiendra cinq propriétés puis le constructeur associé. Vous créez ensuite une méthode *toMap()* qui retournera une *Map* de type *String, dynamic*, en utilisant le même opérateur ternaire pour rendre l'ID nul lorsque sa valeur est 0.

```
class Article {  
    int id;  
    int idListeCourses;  
    String nom;  
    String quantite;  
    String commentaire;  
    Article(this.id, this.idListeCourses, this.nom, this.quantite, this.commentaire);  
    Map<String, dynamic> toMap() {  
        return {  
            'id': (id == 0) ? null : id,  
            'nom': nom,  
            'idCourse': idListeCourses,  
            'quantite': quantite,  
            'commentaire': commentaire,  
        };  
    }  
}
```

Maintenant, dans la classe *GestionBdD*, nous devons créer deux méthodes qui utiliseront les classes de modèle pour insérer des données dans la base de données *courses.db*. Importez les deux classes de modèle dans le fichier *gestion_bdd.dart*. Nous commencerons par la méthode ***insererListeCourses()***, qui insérera un nouvel enregistrement dans le tableau des listes de courses. Comme pour toute opération de base de données, ce sera une fonction asynchrone, et elle retournera un Future de type int, car la méthode d'insertion renverra l'ID de l'enregistrement qui a été inséré. Cette méthode prendra une instance de la classe de modèle *Course* comme paramètre, appelé *course*.

Dans cette méthode, nous appellerons *insert()* de l'objet de base de données : il s'agit d'une méthode d'assistance spécifique exposée par la bibliothèque *sqflite* qui prend trois arguments :

- Le nom de la table où nous voulons insérer des données ('courses' dans ce cas).
- Une Map des données que nous voulons insérer : pour obtenir cela, nous allons appeler notre fonction *toMap()* du paramètre *course*.
- Un paramètre *conflictAlgorithm* qui spécifie le comportement à suivre lorsque vous essayez d'insérer deux fois un enregistrement avec le même ID. Dans ce cas, si la même liste est insérée plusieurs fois, elle remplacera les données précédentes par la nouvelle liste qui a été transmise à la fonction.

Enfin, nous retournons l'id.

```
Future<int> insererListeCourses(ListeCourses listeCourses) async {  
    int id = await this.bdd.insert(  
        'courses',  
        listeCcourse.toMap(),  
        conflictAlgorithm: ConflictAlgorithm.replace,
```

```
    );
    return id;
}
```

Toujours dans la classe *GestionBdD*, vous créez une deuxième méthode, appelée *insererArticle()*, avec exactement le même comportement que la méthode *insererListeCourses()*, qui insérera un article dans la table des articles.

```
Future<int> insererArticle(Article article) async {
    int id = await thisbdd.insert(
        'articles',
        article.toMap(),
        conflictAlgorithm: ConflictAlgorithm.replace,
    );
    return id;
}
```

Nous sommes maintenant prêts à tester ces deux méthodes : nous les appellerons à partir du fichier *main.dart*. En haut du fichier, importons nos deux classes de modèle, puis en haut de la classe *_MyHomePage*, nous allons créer une instance de la classe *GestionBdD*.

Pensez à supprimer le code suivant de la méthode *build* :

```
GestionBdD gestionBdD = GestionBdD();
gestionBdD.testBdD();
```

Vous ajoutez une méthode asynchrone appelée *afficherDonnees()*. Plus tard, cette méthode affichera les données à l'écran, mais pour l'instant, nous allons simplement l'utiliser pour tester nos nouvelles méthodes *insererListeCourses* et *insererArticle*. Dans cette méthode, nous appellerons la méthode *ouvrirBdD()* sur l'objet *gestionBdD* créé en haut de la classe. L'utilisation de la commande *await* garantit que la base de données a été ouverte avant d'essayer d'y insérer des données. Puis, vous créez une instance *ListeCourses* et appelez la méthode *insererListeCourses()*. Nous allons mettre stocker la valeur renournée par *insererListeCourses* dans un entier *idListeCourses*. Vous répétez la même chose pour un Article : ici, l'ID de la liste sera extrait de la variable *idArticle*.

```
Future afficherDonnees() async {
    await gestionBdD.ouvrirBdD();

    ListeCourses listeCourses = ListeCourses(0, 'Boulangerie', 2);
    int idListeCourses = await gestionBdD.insererListeCourses(listeCourses);
    Article article =
        Article(0, idListeCourses, 'Pain', '3 baguettes', 'Bien cuites');
    int idArticle = await gestionBdD.insererArticle(article);

    print('Id course : ' + idListeCourses.toString());
    print('Id article : ' + idArticle.toString());
}
```

Enfin, nous affichons dans la console de débogage les valeurs récupérées, pour vous assurer que tout fonctionne comme prévu.

Il nous reste plus qu'à appeler notre méthode dans la méthode `build()` de la classe `_MyHomePageState`.

```
@override
Widget build(BuildContext context) {
    afficherDonnees();

    return Scaffold(
        appBar: AppBar(
            title: Text(widget.title),
        ),
    );
}
```

Nous sommes maintenant prêts à exécuter l'application. Si tout a fonctionné comme prévu, vous devriez maintenant voir les ID des enregistrements que nous avons insérés dans la base de données (les nombres que vous voyez peuvent varier en fonction du nombre d'enregistrements dans votre base de données).



```
PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL
Launching lib\main.dart on sdk gphone x86 64 arm64 in debug mode...
✓ Built build\app\outputs\flutter-apk\app-debug.apk.
Connecting to VM Service at ws://127.0.0.1:62874/M8sdooSrGPw=/ws
I/flutter (27529): Id course : 1
I/flutter (27529): Id article : 1
```

Bravo ! Nous avons créé les classes de modèle qui faciliteront la gestion de la base de données et utilisé la méthode d'insertion pour insérer des données dans nos tables. Il est maintenant temps de montrer ces données à notre utilisateur.

7.3.4 Affichage des données de la base de données à l'utilisateur

Maintenant que nous avons ajouté des données dans notre base de données `courses.db`, il est temps de montrer ces données à l'utilisateur. Nous commencerons par afficher les listes de courses disponibles sur le premier écran, dans un `ListView`. Une fois que l'utilisateur a appuyé sur un élément de la liste, il accédera au deuxième écran de l'application, qui affichera tous les articles de la liste d'achats.

Tout d'abord, nous allons créer une fonction qui récupère le contenu de la table `courses` dans notre base de données, en utilisant une méthode de `sqflite`. Pour cela, dans la classe `GestionBdD`, ajoutez une nouvelle méthode appelée `lireListesCourses()` qui retournera un `Future<List>`, contenant une liste des listes de courses. Comme d'habitude, cette méthode sera asynchrone.

Dans la fonction, appelez la méthode `query` sur la base de données. Comme celle-ci récupérera toutes les données de la table des courses, le seul paramètre requis ici est le nom de la table.

```

Future<List<Course>> lireListesCourses() async {
    final List<Map<String, dynamic>> maps = await this.bdd.query('courses');
    return List.generate(maps.length, (i) {
        return ListeCourses(
            maps[i]['id'],
            maps[i]['nom'],
            maps[i]['priorite'],
        );
    });
}

```

Notez que la méthode `query()` renvoie une liste d'éléments de type `Map`. Afin de les utiliser plus facilement, nous devons convertir la `List<Map<String, dynamic>` en `List<ListeCourses>`. Nous pouvons le faire en appelant la méthode `List.generate()`, que vous pouvez utiliser pour générer une liste de valeurs. Le premier paramètre spécifie la taille de la liste et le second est une fonction qui génère les valeurs de la liste. La valeur de retour ici est une liste d'objets `ListeCourses`, ce que nous voulions obtenir. Une fois que nous avons la liste des objets `ListeCourses`, nous devons les afficher sur le premier écran de notre application. Dans le fichier `main.dart`, en haut de la classe `_MyHomePageState`, créez une propriété `List<ListeCourses>` qui sera une liste de courses.

Dans la méthode `afficherDonnees()`, supprimez tout le code de test à l'exception de l'ouverture de la base de données. Sous cette ligne, nous appellerons la fonction `lireListesCourses()` de notre objet `gestionBdD`. Puis nous appelerons la méthode `setState()` pour indiquer à notre application que la liste des courses a changé.

```

Future afficherDonnees() async {
    await gestionBdD.ouvrirBdD();
    listesCourses = await gestionBdD.lireListesCourses();
    setState(() {
        listesCourses = listesCourses ;
    });
}

```

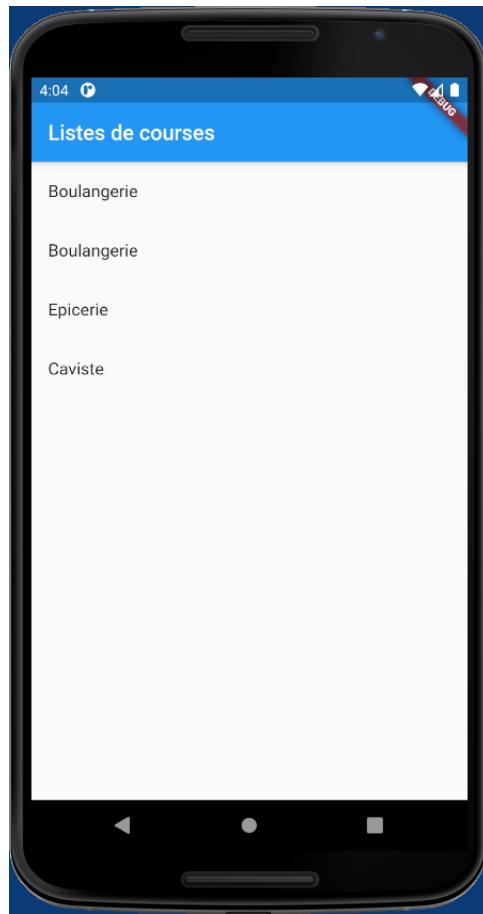
Maintenant que nous avons récupéré toutes les données nécessaires, nous devons les afficher sur l'écran. Dans la méthode `build()` de la classe `_MyHomePageState`, dans la priorité `body` du `Scaffold`, utilisez un `ListView.builder`, qui contiendra le nombre d'éléments disponibles dans la variable `listesCourses`. Si `listesCourses` est nul, alors le `itemCount` de `ListView` sera 0. Pour ce faire, comme d'habitude, nous utiliserons la syntaxe d'opérateur ternaire. Dans `itemBuilder`, retournez un `ListTile` dont le titre sera la propriété `nom` de la liste de courses, à l'index de position.

```

body: ListView.builder(
    itemCount: (courses != null) ? courses.length : 0,
    itemBuilder: (BuildContext context, int index) {
        return ListTile(
            title: Text(listesCourses[index].nom),
        );
    },
),

```

Si tout a fonctionné comme prévu, si vous essayez l'application maintenant, vous devriez voir une liste des listes de courses que nous avons insérées jusqu'à présent.



Si vous n'avez pas ou peu de données, vous pouvez créer une fonction `initialiserDonnesBdd()` qui retournera un Future en mode asynchrone.

```
Future initialiserDonneesBdd() async {
    await gestionBdD.ouvrirBdD();

    ListeCourses listeCourses1 = ListeCourses(0, 'Boulangerie', 2);
    int idListeCourses1 = await gestionBdD.insererListeCourses(listeCourses1);
    Article article10 =
        Article(0, idListeCourses1, 'Pain', '3 baguettes', 'Bien cuites');
    await gestionBdD.insererArticle(article10);
    Article article11 =
        Article(0, idListeCourses1, 'Croissant', '10 pièces', 'Au beurre');
    await gestionBdD.insererArticle(article11);
    Article article12 = Article(
        0, idListeCourses1, 'Chocolatine', '15 pièces', 'Beaucoup de chocolat');
    await gestionBdD.insererArticle(article12);

    ListeCourses listeCourses2 = ListeCourses(0, 'Epicerie', 3);
    int idListeCourses2 = await gestionBdD.insererListeCourses(listeCourses2);
    Article article20 =
```

```

    Article(0, idListeCourses2, 'Fruits', '2 kg', 'Pommes rouges');
    await gestionBdD.insererArticle(article20);
    Article article21 =
        Article(0, idListeCourses2, 'Pâtes', '500 g', 'Coquillettes');
    await gestionBdD.insererArticle(article21);

    ListeCourses listeCourses3 = ListeCourses(0, 'Caviste', 1);
    int idListeCourses3 = await gestionBdD.insererListeCourses(listeCourses3);
    Article article30 =
        Article(0, idListeCourses3, 'Bières', '3 fûts', 'Leffe Royale');
    await gestionBdD.insererArticle(article30);
    Article article31 = Article(
        0, idListeCourses3, 'Paix Dieu', '5 bouteilles', '75 cl avec 6 verres');
    await gestionBdD.insererArticle(article31);
}

```

Cette fonction insérera des données dans votre base de données en l'appelant dans la méthode `initState` de votre classe `_MyHomePageState`.

```

@Override
void initState() {
    initialiserDonneesBdd();
    super.initState();
}

```

Afin de compléter cet écran, nous ajouterons quelques données supplémentaires. Tout d'abord, pour rendre l'interface utilisateur un peu plus attrayante, ajoutez un `CircleAvatar` à la propriété principale `ListTile`, contenant la priorité. Ensuite, toujours dans le `ListTile`, ajoutez une icône en bout de ligne que nous utiliserons plus tard pour éditer la liste de courses.

```

return ListTile(
    title: Text(listesCourses[index].nom),
    leading: CircleAvatar(
        child: Text(listesCourses[index].priorite.toString())),
    ),
    trailing: IconButton(
        icon: Icon(Icons.edit),
        onPressed: () {}),
),
);

```

Maintenant que nous pouvons voir la liste d'achats, montrons également les articles sur chaque liste. Pour cela, nous aurons besoin d'un nouveau fichier. Afin de mieux organiser notre code, créez un nouveau dossier appelé « `pages` » qui contiendra les fichiers des pages interface utilisateur de nos projets, sauf `main.dart`, qui restera dans le dossier `lib`. Dans le dossier `pages`, créez un nouveau fichier appelé `page_articles.dart`.

Dans ce nouveau fichier, importez d'abord les fichiers dont nous aurons besoin pour afficher les éléments. Créez ensuite un *stateful* widget, appelé ***PageArticles***. Chaque fois que nous arriverons à cet écran, ce sera parce que nous avons sélectionné une instance *ListeCourses*. Nous n'aurons jamais besoin d'appeler cet écran indépendamment. Ainsi, il est logique que lorsque nous créons le widget *PageArticle*, nous nous attendions à ce qu'une instance de *ListeCourses* soit transmise. De ce fait, vous créez en haut de la classe *PageArticles* une variable *listeCourses*, et ajoutez également un constructeur qui définira la propriété *listeCourses*. Ensuite vous transmettez cette propriété à la classe *_PageArticlesState*.

```
import 'package:flutter/material.dart';
import '../modeles/liste_courses.dart';
import '../modeles/article.dart';
import '../util/gestion_bdd.dart';

class PageArticles extends StatefulWidget {
    final ListeCourses listeCourses;
    PageArticles(this.listeCourses);
    @override
    _PageArticlesState createState() => _PageArticlesState(this.listeCourses);
}

class _PageArticlesState extends State<PageArticles> {
    final ListeCourses listeCourses;
    _PageArticlesState(this.listeCourses);
    @override
    Widget build(BuildContext context) {
        return Container();
    }
}
```

Maintenant que nous avons défini *listeCourses*, dans la méthode *build()* de la classe *_PageArticlesState*, retournez un *Scaffold* qui, dans le titre *AppBar*, affiche le nom de la liste de course. Afin de tester l'application, appelez *PageArticles* lorsque l'utilisateur appuie sur l'un des éléments de la *ListView* de l'écran principal dans le fichier *main.dart*. Pour cela, nous allons d'abord importer le fichier *page_articles.dart*, puis dans la méthode *build()* de la classe *_MyHomePageState*, à l'intérieur du *ListTile* de *ListView*, ajoutez un paramètre *onTap*. À l'intérieur,appelez la méthode *Navigator.push()* pour appeler *PageArticles*, en passant l'objet dans *listeCourse* à l'index de position.

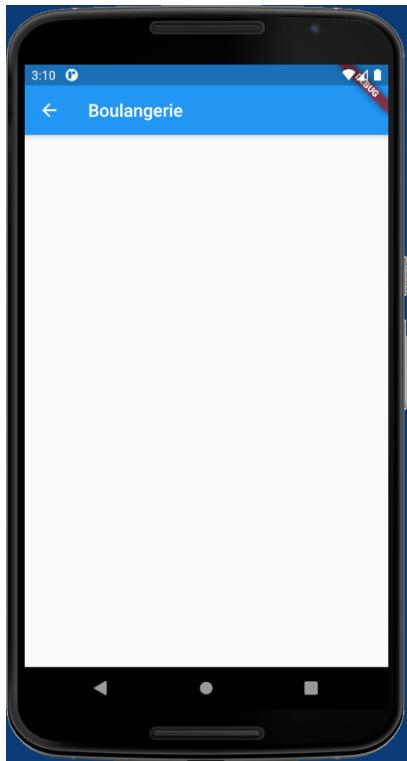
```
onTap: () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => PageArticles(listesCourses[index])),
    );
},
```

Si vous essayez ceci, lorsque vous appuyez sur l'un des éléments de la ListView, vous arriverez au deuxième écran, qui pour l'instant n'affiche que le titre de la liste de courses, comme indiqué dans la capture d'écran ci-contre.

La prochaine étape est d'afficher la liste dans le deuxième écran, qui contiendra les articles de la liste d'achats sélectionnés à partir du premier écran. Par conséquent, nous devons créer une méthode qui interroge la base de données dans la table des éléments, en passant l'ID de la liste de courses qui a été sélectionnée et qui renvoie tous les éléments récupérés. Nous ajouterons cette méthode dans la classe GestionBdD, avec toutes les autres méthodes qui traitent de la base de données. Comme d'habitude, ce sera une méthode asynchrone qui retournera un Future de type *List<Article>*.

Ensuite, comme nous l'avons fait pour la méthode *lireListesCourses()*, nous allons appeler la méthode *query* sur la base de données, en passant le nom de la table, *articles*, comme premier argument. Mais nous allons également définir un deuxième argument, nommé *where*, qui filtrera les résultats en fonction d'un champ spécifique - dans notre cas, *idCourses*. La variable *idCourses* sera égale à la valeur que nous définirons dans le paramètre nommé *whereArgs*. Dans ce cas, l' *idCourses* devra être égal à la valeur qui a été passée à la fonction *lireArticles()*. Comme vous vous en souvenez peut-être, cela renverra une *List<Map<String,dynamic>*. Nous placerons le résultat de la requête sur une variable appelée *maps*, puis nous la convertirons en *List<Map<String,dynamic>* en *List<Article>* avant de la retourner.

```
Future<List<Article>> lireArticles(int idListeCourses) async {
    final List<Map<String, dynamic>> maps = await this.bdd.query(
        'articles',
        where: 'idCourses = ?',
        whereArgs: [idListeCourses],
    );
    return List.generate(maps.length, (i) {
        return Article(
            maps[i]['id'],
            maps[i]['idCourses'],
            maps[i]['nom'],
            maps[i]['quantite'],
            maps[i]['commentaire'],
        );
    });
}
```



Il est maintenant temps d'afficher les éléments à l'utilisateur. Dans le fichier *page_articles.dart*, en haut de la classe *_PageArticlesState*, nous allons créer deux propriétés: l'une sera le gestionnaire de la base de données, et une autre contiendra tous les articles à afficher.

Maintenant, si vous y réfléchissez, nous n'avons pas besoin d'avoir plusieurs instances de la classe `GestionBdD` dans toute l'application. Avoir une seule connexion à la base de données est ce dont nous avons réellement besoin. Dans Dart et Flutter, il existe une fonctionnalité appelée «constructeurs d'usine» notée **factory** qui remplace le comportement par défaut lorsque vous appelez le constructeur d'une classe. Au lieu de créer une nouvelle instance, le constructeur d'usine ne renvoie qu'une instance de la classe. Dans notre cas, cela signifie que la première fois que le constructeur d'usine est appelé, il renverra une nouvelle instance de `GestionBdD`. Une fois que `GestionBdD` a déjà été instancié, le constructeur ne construira pas une autre instance, mais retournera simplement celle existante. Pour cela, nous ajoutons le code ci-dessous à la classe `GestionBdD`.

```
class GestionBdD {  
    ...  
    static final GestionBdD _gestionBdD = GestionBdD._internal();  
    GestionBdD._internal();  
    factory GestionBdD() {  
        return _gestionBdD;  
    }  
    ...  
}
```

En détail, tout d'abord, nous créons un constructeur privé nommé `_internal`. Ensuite, dans le constructeur d'usine, nous le renvoyons simplement à l'appelant extérieur.

Dans le fichier `page_articles.dart` de la classe `_PageArticlesState`, nous allons créer une méthode asynchrone nommée `afficherArticles()` qui prendra l'ID de la liste de courses qui a été passé à la classe. Dans le haut de la classe, nous définissons deux variables `gestionBdD` et `articles` qui est une liste d'`article`.

```
GestionBdD gestionBdD = GestionBdD();  
List<Article> articles;
```

Dans la méthode `afficherArticles`,appelez d'abord la méthode `ouvrirBdD()` pour vous assurer que la base de données est disponible et ouverte, puis la méthode `lireArticles()` de l'objet `GestionBdD` passant `idListeCourses`. Le résultat de la méthode `lireArticles()` sera placé dans la propriété `articles`. Ensuite, appelez la méthode `setState()` pour mettre à jour la propriété `State`, afin que l'interface utilisateur soit redessinée.

```
Future afficherArticles(int idListeCourses) async {  
    await gestionBdD.ouvrirBdD();  
    articles = await gestionBdD.lireArticles(idListeCourses);  
    setState(() {  
        articles = articles;  
    });  
}
```

8 Accéder au WEB depuis une application Flutter

8.1 Introduction

Dans ce chapitre, nous allons voir comment récupérer des données à partir d'un service Web. Vous verrez les méthodes HTTP et le stockage des données couvrant les fonctionnalités de base de la plupart des applications professionnelles. Ces applications impliquent une communication HTTP entre votre application et certaines API sur certains serveurs. La plupart du temps, ces API de serveur sont construites selon les directives de conception REST et les données seront transférées au format JSON. Le but de ce paragraphe est un rappel sur HTTP, les API, REST et JSON que vous avez abordé du coté serveur.

Lorsque votre application communique vers et depuis un serveur distant via HTTP, elle le fait de manière asynchrone. L'application ne s'arrête pas soudainement complètement après avoir envoyé une requête au serveur. Comme vous l'avez déjà vu précédemment, le langage Dart prend entièrement en charge la programmation asynchrone, y compris *Futures*.

Le package HTTP Flutter utilise *Futures* pour permettre aux développeurs de communiquer via HTTP de manière asynchrone. Chaque fois que nous communiquons avec le serveur via HTTP, nous n'arrêtions pas de faire des choses dans l'application, mais nous traitons la réponse de succès ou d'erreur lorsqu'elle nous revient.

8.2 Rappel http

Le protocole HTTP (Hypertext Transfer Protocol) est conçu pour permettre les communications entre les clients et les serveurs. HTTP fonctionne comme un protocole de question-réponse entre un client et un serveur. Un protocole décrit comment les machines communiquent entre elles à l'aide de messages. Un protocole définit le format de ces messages.

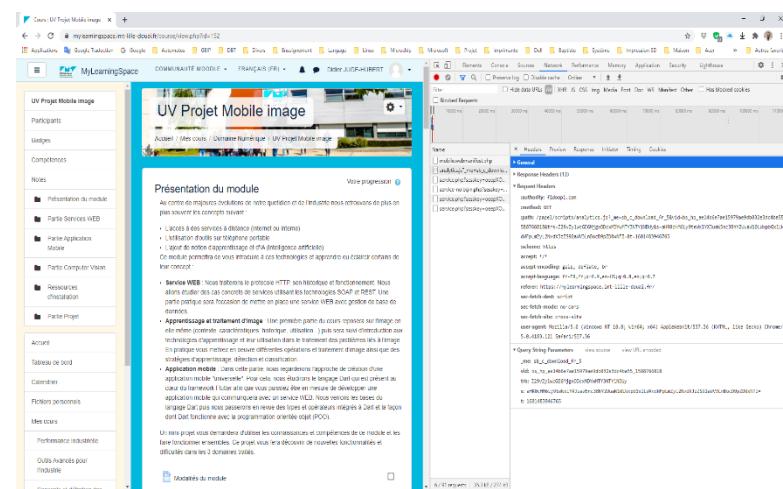
Les réponses renvoyées par le serveur disposent d'un status qui est codifié. Il indique si la demande a été traitée avec succès ou non. Voici quelques-unes des valeurs de code d'état http :

Code	Désignation	Description
1xx	<i>Informational</i>	
2xx	<i>Success</i>	
	200 <i>Ok</i>	
3xx	<i>Redirect</i>	
	301 <i>Moved permanently</i>	
	302 <i>Moved temporarily</i>	
4xx	<i>Request error</i>	
	400 <i>Bad request</i>	La demande n'a pas pu être comprise par le serveur.
	403 <i>Forbidden</i>	Utilisateur non autorisé à effectuer l'opération demandée.
	404 <i>Not found</i>	La ressource demandée est introuvable à l'URI donné.
	405 <i>Method not allowed</i>	La méthode de demande n'est pas autorisée sur la ressource spécifiée.
5xx	<i>Server error</i>	
	500 <i>Internal server error</i>	Le serveur a rencontré une condition inattendue, l'empêchant de répondre à la demande.
	503 <i>Service unavailable</i>	Le serveur est temporairement indisponible, généralement en raison d'une surcharge ou d'une maintenance.

8.2.1 Outils

Vous commencez à maîtriser Flutter, vous finirez par passer beaucoup de temps à écrire du code qui communique avec les serveurs utilisant HTTP. Etude de ces outils à l'avance vous facilitera la vie.

Vous en connaissez évidemment déjà un. Si vous voulez voir le protocole HTTP au travail, ouvrez votre navigateur, allez sur un site Web puis utilisez le menu permettant d'accéder aux outils de développement.

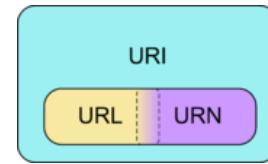


Selectionnez l'option « **Network** » pour afficher l'inspecteur du trafic réseau. Dans l'image ci-contre, vous pouvez voir l'inspecteur du trafic réseau sur le côté droit, avec une demande sélectionnée et vue plus en détail.

Un autre outil Postman, téléchargeable à l'adresse suivante : <https://www.postman.com/downloads/> vous permettra de tester les requêtes HTTP vers un serveur avant de les coder dans Flutter. Avec ce dernier outil, vous pourrez afficher les données brutes et voir ce qui se passe.

8.2.2 L'identification du serveur : URI (Uniform Resource Identifier)

Il s'agit de l'adresse de destination de la demande, soit un chemin spécifique sur un serveur spécifique (ex : <https://mylearningspace.imt-lille-douai.fr/>).



Cette URI peut contenir des informations visibles ou non, principalement au travers de paramètres. HTTP vous permet de transmettre des informations au serveur dans l'URL en utilisant des paramètres de requête (Ex : <http://localhost:4200/sockjs-node/info?t=1498649243238>). En outre, le chemin de l'URI (URL : Uniform Ressource Locator) est un lui-même un paramètre.

Certains caractères ne peuvent pas faire partie d'une URL (par exemple des espaces) et certains autres caractères ont une signification particulière dans une URL. Pour contourner ce problème, la syntaxe de l'URL permet le codage des paramètres pour garantir une URL valide (Ex : Le caractère « espace » sera codé en « %20 » dans une URL).

8.2.3 Les trames HTTP

Les trames HTTP sont constituées de plusieurs parties :

Header (entête) : Les entêtes HTTP permettent au client et au serveur de transmettre des informations supplémentaires avec la requête ou la réponse. Un entête de demande se compose de paires clé/valeur - une clé insensible à la casse suivie de deux points « :: », puis de sa valeur (sans saut de ligne).

Body (corps de la trame) : Le corps HTTP permet au client et au serveur de transmettre des informations supplémentaires avec la requête ou la réponse après l'en-tête.

Dans la requête, les corps HTTP ne sont pas toujours requis car un corps d'informations n'est pas toujours nécessaire. Les requêtes HTTP GET et DELETE n'ont généralement pas besoin de corps, par contre, les requêtes HTTP POST, PUT et PATCH ont un corps. Ce dernier contient les informations à créer ou à modifier qui sont envoyées.

Pour la réponse, le corps est utilisé pour renvoyer des informations dans la réponse et il peut devenir très volumineux, avec une quantité considérable de données.

La requête est constituée de méthodes HTTP qui existent depuis longtemps. Les méthodes HTTP les plus couramment utilisées sont POST, GET, PUT, PATCH et DELETE.

La « méthode » décrit ce que l'application souhaite que le serveur fasse, quelle est l'intention de la demande. Les méthodes les plus couramment utilisées sont **get** et **post**. La méthode **get** est utilisée pour demander des données au serveur. La méthode **post** est utilisée pour envoyer des données au serveur, pour les sauvegarder ou les mettre à jour. La méthode **put** est utilisée pour mettre à jour les données sur le serveur. La méthode **delete** est utilisée pour supprimer des données sur le serveur.

8.2.4 Les API

Lorsque quelqu'un met son API à la disposition du monde entier, il écrit le code de l'API et le publie sur son serveur Web HTTP. Les API sont également appelées services Web. La plupart des API utilisent le style architectural REST, qui est un modèle de la façon dont vous communiquerez avec le serveur via HTTP. Les API conformes au style architectural REST fonctionnent principalement de la même manière, avec des adresses Web (URI) et des méthodes HTTP similaires.

8.2.5 REST (Representational State Transfer).

REST nous donne des directives de conception de haut niveau et vous laisse penser à votre propre implémentation. Les API REST doivent être sans état.

Dans le passé, les applications Web permettaient de stocker les données de session de l'utilisateur. Par exemple, l'utilisateur se connecterait et cela lancerait une session et les informations pourraient être conservées dans cette session jusqu'à ce que l'utilisateur se déconnecte. Ces données de session peuvent inclure qui est l'utilisateur, quel accès il a et toute autre information requise.

Désormais, avec des API REST plus modernes, l'accès aux serveurs est contrôlé via des jetons (*token*) ou des clés (*key*) API. En outre, chaque appel d'API est sans état - chaque demande du client au serveur est autonome et contient toutes les données permettant d'identifier qui a effectué la demande et toutes les données de la demande elles-mêmes pour effectuer l'opération. Une telle demande ne peut tirer parti d'aucune donnée de session préexistante sur le serveur.

8.2.5.1 Identification de l'utilisateur par jeton

Dans la plupart des applications avec une connexion, lorsqu'une connexion utilisateur se produit, il ou elle reçoit un jeton temporaire pour l'accès. Ce jeton est chiffré et contient des informations sur l'utilisateur et le jeton lui-même (par exemple, quand il expire). Ce jeton peut être actualisé à chaque période de temps prédéterminée (par exemple toutes les 15 minutes).

Chaque fois qu'un appel à l'API est effectué à partir d'un appareil vers le serveur, le jeton doit être inclus dans chaque en-tête de demande sortante vers le serveur. Si le jeton n'est pas présent ou invalide (il peut expirer), le serveur renvoie un code d'erreur (généralement un code HTTP 401 ou 403). Si le jeton est bon, le serveur sait qu'un utilisateur connecté valide utilise l'application, le serveur dispose d'informations sur l'utilisateur à partir du jeton et l'API peut effectuer son opération.

8.2.5.2 Identification par clé d'API

Si l'utilisateur n'a pas vraiment besoin de se connecter à chaque fois que l'application est utilisée, une clé d'API permet à un utilisateur enregistré d'être identifié dans l'en-tête HTTP en tant qu'utilisateur valide pour chaque requête sortante du serveur. Comme un jeton, cela est vérifié et le serveur renvoie un code d'erreur en cas de problème.

8.2.5.3 Comment REST utilise les URL

Dans REST, l'URL est utilisée pour déterminer à quelle ressource vous le faites. Elle est constitué :

- ⊕ **URL de base** : C'est la première partie de l'API, sans la partie REST. La partie REST vient après l'URL de base. L'URL de base est généralement la suivante :
 - Le domaine. Par exemple. www.example.com.
 - Éventuellement, un suffixe « api » pour indiquer que le chemin est réservé à l'utilisation de l'API.
 - Éventuellement, un suffixe pour le nom de l'application pour laquelle l'API a été écrite.
 - En option, il a également la version API.
- ⊕ Chemin d'accès : Pensez à URL comme chemin vers la ressource (les données).
<http://www.example.com/customers/33245/orders/8769/lineitems/1>
 - *Doit être considéré comme* :
 - *Allez au client 33245.*
 - *Ensuite, passez à la commande 8769 pour ce client.*
 - *Ensuite, accédez à l'élément 1 pour cette commande.*

8.2.5.4 Comment REST utilise les méthodes HTTP

Dans REST, les méthodes HTTP sont utilisées pour décrire ce que vous faites : obtenir des données, publier de nouvelles données, les mettre à jour, les supprimer.

8.2.5.4.1 Pour obtenir des données avec API REST

Dans ce cas,

- ⊕ URL
 - Identification des données auxquelles vous voulez accéder.
 - Une liste d'articles.
 - URL de base + le nom de la ressource : Dans ce cas, elle renverrait généralement plusieurs projets ou liste d'éléments pouvant appartenir à une autre entité.
 - Une liste d'articles recherchés.
 - L'URL serait similaire à la liste des éléments ci-dessus, plus quelques informations supplémentaires à la fin pour spécifier la recherche.
 - Information additionnelle.
 - Vous pouvez ajouter des chaînes de requête ou des paramètres de requêtes à la fin de l'URL.
- ⊕ Méthode HTTP
 - Vous devez utiliser une méthode HTTP « **get** » pour accéder aux données via une API REST.
- ⊕ Corps HTTP
 - Non utilisé.

8.2.5.4.2 Pour insérer des données avec API REST

Dans ce cas,

- ⊕ URL
 - Identification du type de données que vous insérez.
 - Ce serait la même que l'URL de la liste des éléments.
- ⊕ Méthode HTTP

- Vous devez utiliser une méthode HTTP « **put** » pour insérer (ou créer) des données via une API REST.
- + Corps HTTP
 - Vous placez normalement les données requises pour l'insertion dans le corps de la requête.

8.2.5.4.3 Pour mettre à jour des données avec API REST

Dans ce cas,

- + URL
 - Identifie les données que vous mettez à jour.
- + Méthode HTTP
 - Vous utilisez une méthode HTTP « **put** » pour mettre à jour les données via une API REST.
- + Corps HTTP
 - Vous placez normalement les données requises pour la mise à jour dans le corps de la requête.

8.2.5.4.4 Pour supprimer des données avec API REST

Dans ce cas,

- + URL
 - Identifie les données que vous supprimez.
- + Méthode HTTP
 - Vous devez utiliser une méthode HTTP « **delete** » pour supprimer des données via une API REST .
- + Corps HTTP
 - Non utilisé.

8.2.6 Le protocole JSON (JavaScript Object Notation)

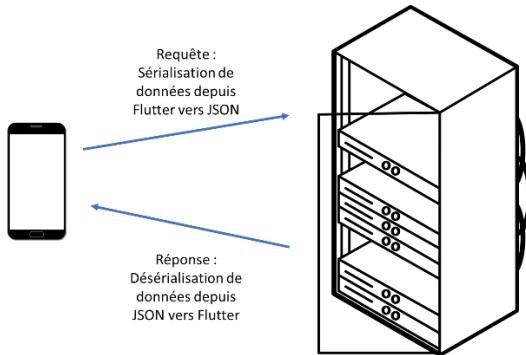
C'est un format de données utilisé pour transmettre des données entre le client et le serveur (dans les deux sens). Il s'agit du même format de données utilisé par le langage JavaScript. Il utilise une virgule pour séparer les éléments et un signe deux-points pour séparer le nom d'une propriété avec les données de cette propriété. Il utilise différents types pour désigner les objets {} et les tableaux []. Il est possible de mixer les deux pour transmettre des tableaux d'objets.

8.3 Flutter avec HTTP, API, REST et JSON

Le but de ce paragraphe est d'écrire du code Flutter qui communique avec les API via HTTP en utilisant REST avec JSON comme format de données.

8.3.1 Flutter & JSON

Nous savons donc que nous communiquons avec les serveurs en utilisant le protocole HTTP, en utilisant JSON comme format de données.



Requête : Lorsque l'application envoie une demande sortante à une API sur un serveur, elle doit souvent convertir des données Flutter (par exemple des données dans un formulaire) en JSON. Cette conversion des données Flutter en données JSON s'appelle la sérialisation.

Réponse : Lorsque le serveur répond, l'application doit convertir les données JSON en données Flutter. Cette conversion des données JSON en données Flutter s'appelle la désérialisation.

8.3.2 Sérialisation et désérialisation de JSON

Nous savons donc que nous devons convertir les données entre le JSON et Flutter.

- ⊕ Par JSON, nous entendons une chaîne de caractères.
- ⊕ Par Flutter, nous entendons "données dans une classe Dart dans notre application Flutter".

Les deux principales méthodes de sérialisation et de désérialisation de JSON dans une application Flutter :

Automatique => Utilisation d'un package de génération de code pour la sérialisation et la désérialisation

- *Avantages* : Vous n'avez pas besoin d'écrire le code. Le code généré, il ne fait pas d'erreur.
- *Inconvénients* : Ce n'est pas très simple à configurer, vous devez savoir comment cela fonctionne. Cela ne fonctionne pas avec les cas compliqués pour les coder.

Manuel => Écrire manuellement du code pour la sérialisation et la désérialisation

- *Avantages* : Vous devez écrire le code. Vous pouvez coder les scénarios de sérialisation et de désérialisation les plus complexes.
- *Inconvénients* : Il y aura des bugs. Ce n'est pas très simple à coder, vous devez savoir comment cela fonctionne.

Vous pouvez combiner les deux. Par exemple appliquer la règle des 80/20, c'est-à-dire :

- Faire 80% en mode automatique : C'est la méthode la plus simple, en générant le code pour la sérialisation et la désérialisation d'objets simples.
- Faire 20% en mode manuel : C'est plus compliqué car vous devez créer votre propre code pour sérialiser et désérialiser des objets plus complexes.

Les exemples de code donnés ci-dessous utilise cette règle. Nous faisons les choses faciles en utilisant le générateur de code (sérialisation et désérialisation simples) et les choses difficiles (sérialisation et désérialisation récursives) dans le code manuscrit.

8.3.2.1 Génération de code pour la sérialisation et la désérialisation

Cette approche utilise deux packages : Le package **json_serializable** pour générer le code de sérialisation et de désérialisation et le package **build_runner** qui fonctionne avec le package « json_serializable » génère les fichiers de code.

Dans la **première étape**, nous devons ajouter des dépendances aux projets en modifiant le fichier de dépendances du projet *pubspec.yaml* comme suit puis le sauvegarder afin de récupérer les packages :

```
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^3.0.0

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter
  build_runner: ^1.0.0
  json_serializable: ^3.2.0
  json_serializable: ^3.5.0
```

La **deuxième étape** est de définir les classes à sérialiser et désérialiser. Pour cela, dans un nouveau fichier (ex : *person.dart*), il faut importer les annotations JSON, intégrer le fichier de sérialisation « *.g.dart » (ex : *person.g.dart*) et enfin créer la classe (ex : *Person*) en indiquant les champs JSON correspondant si nécessaire.

```
import 'package:json_annotation/json_annotation.dart';

part 'person.g.dart';

@JsonSerializable()
class Person {
  final String name;
  @JsonKey(name: "addr1")
  final String addressLine1;
  @JsonKey(name: "city")
  final String addressCity;
  @JsonKey(name: "state")
  final String addressState;

  Person(this.name, this.addressLine1, this.addressCity, this.addressState);

  @override String toString() {
    return 'Person{name: $name, addressLine1: $addressLine1, addressCity: $addressCity, addressState: $addressState}';
  }
}
```

```

factory Person.fromJson(Map<String, dynamic> json) => _$PersonFromJson(json)
;

Map<String, dynamic> toJson() => _$PersonToJson(this);
}

```

Pour la classe, nous avons ajoutez une annotation `@JsonSerializable()` juste avant la déclaration de la classe. Puis nous avons ajoutez les annotations de champ JSON juste avant les déclarations des propriétés.

Celles-ci ne sont pas nécessaires si le nom du champ JSON reste le même que le nom de la propriété. L'annotation `@JsonProperty` déclare le nom JSON du champ si vous souhaitez qu'il soit différent du nom de la propriété.

Pour la **troisième étape**, nous devons maintenant générer le fichier de code de sérialisation et de désérialisation ayant extension «`.g.dart`» que nous avons intégrer à notre fichier de base. Pour cela, nous devons exécuter la ligne de commande suivante à la racine du projet :

```
Flutter packages pub run build_runner build
```

Cela devrait générer un fichier «`.g.dart`» dans le projet pour chaque fichier où vous avez ajouté l'annotation `@JsonSerializable`.

```

// GENERATED CODE - DO NOT MODIFY BY HAND

part of 'person.dart';

// *****
// JsonSerializableGenerator
// *****

Person _$PersonFromJson(Map<String, dynamic> json) {
    return Person(
        json['name'] as String,
        json['addr1'] as String,
        json['city'] as String,
        json['state'] as String,
    );
}

Map<String, dynamic> _$PersonToJson(Person instance) => <String, dynamic>{
    'name': instance.name,
    'addr1': instance.addressLine1,
    'city': instance.addressCity,
    'state': instance.addressState,
};

```

Notez que ce fichier contient les méthodes Dart qui ont été incorporées dans d'autres classes sans utiliser d'héritage. Dans l'exemple, cela génère un fichier *person.g.dart* pour correspondre au fichier *person.dart*

REMARQUE : Assurez-vous de réexécuter la commande suivante dans la racine de votre projet chaque fois que vous modifiez quelque chose :

```
Flutter packages pub run build_runner build
```

Un premier exemple d'utilisation de cette classe vous est donné ci-dessous dans le projet « *serialize_with_generated_code* ».

Cette application crée une instance *_person* de la classe Person et affiche *_person.toString()* en noir.

Ensuite, elle sérialise également l'instance précédente et affiche le résultat JSON en dessous en rouge.

Il existe un bouton "Copy" pour copier le JSON dans le presse-papiers afin de pouvoir le coller dans un formateur JSON en ligne. N'oubliez pas que cela ne devrait pas fonctionner de manière récursive, contrairement à l'exemple avec le code écrit manuellement.

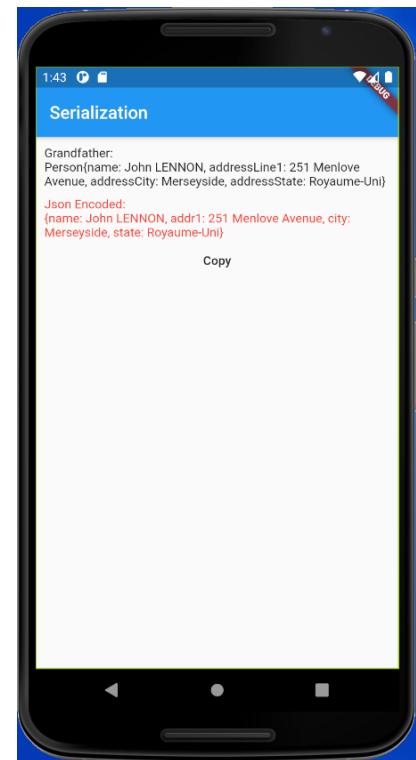
```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

import 'person.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'JSON serialization',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Serialization'),
    );
  }
}

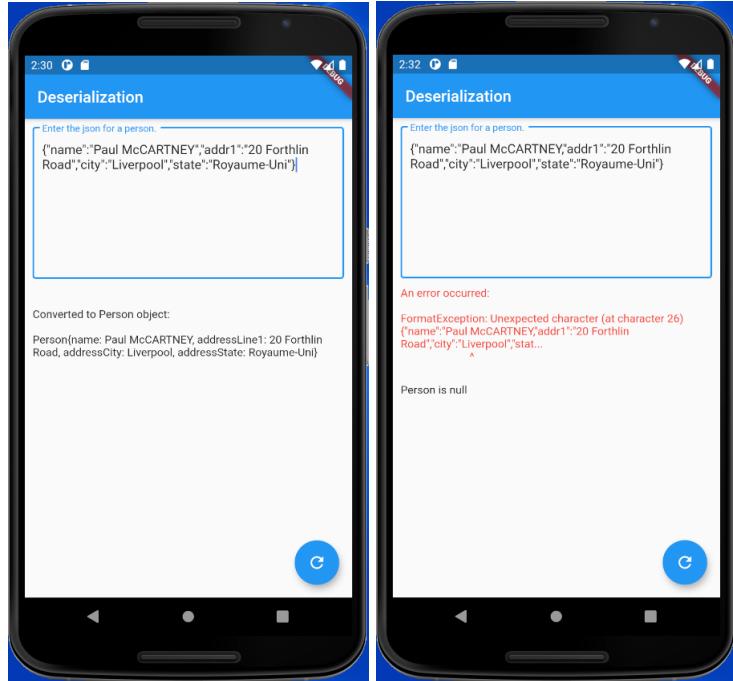
class MyHomePage extends StatelessWidget {
```



```
MyHomePage({Key key, this.title}) : super(key: key);
final String title;
static Person _person = Person(
    "John LENNON",
    "251 Menlove Avenue",
    "Merseyside",
    "Royaume-Uni");
@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text(title),
        ),
        body: Center(
            child: Padding(
                child: ListView(
                    children: <Widget>[
                        Padding(
                            child: Text("Grandfather:\n${_person}"),
                            padding: EdgeInsets.only(top: 0.0),
                        ),
                        Padding(
                            child: Text(
                                "Json Encoded:\n${_person.toJson()}",
                                style: TextStyle(color: Colors.red),
                            ),
                            padding: EdgeInsets.only(top: 10.0),
                        ),
                        FlatButton(
                            child: Text('Copy'),
                            onPressed: () {
                                Clipboard.setData(ClipboardData(text: "${_person.toJson()}"));
                            },
                        ),
                    ],
                ),
                padding: EdgeInsets.all(10.0),
            ),
        );
    }
}
```

Un deuxième exemple vous est donné ci-dessous dans l'application « deserialize_with_generated_code ». Cette application vous permet de saisir le JSON d'une personne, puis d'appuyer sur le bouton pour le déserialiser. En cas de succès, la personne est affiché au travers de la méthode `toString()` de l'objet `Person` en dessous (en noir). Si une erreur se produit (peut-être avez-vous entré un mauvais JSON?), elle est affichée en dessous (en rouge).

N'oubliez pas que cela ne devrait pas fonctionner de manière récursive, contrairement à l'exemple avec le code écrit manuellement.



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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'JSON Deserialize',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
      ),
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key}) : super(key: key);
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

```

class _MyHomePageState extends State<MyHomePage> {
  final _jsonTextController = TextEditingController();
  Person _person;
  String _error;

  _MyHomePageState() {
    final String person =
        "{\"name\":\"Paul McCARTNEY\",\"addr1\":\"20 Forthlin Road\",\"city\":
\"Liverpool\",\"state\":\"Royaume-Uni\"}";
    _jsonTextController.text = person;
  }

  TextFormField _createJsonTextField() {
    return new TextFormField(
      validator: (value) {
        if (value.isEmpty) {
          return 'Please enter the json.';
        }
      },
      decoration: InputDecoration(
        border: OutlineInputBorder(),
        hintText: 'Json',
        labelText: 'Enter the json for a person.'),
      controller: _jsonTextController,
      autofocus: true,
      maxLines: 8,
      keyboardType: TextInputType.multiline,
    );
  }

  _convertJsonToPerson() {
    _error = null;
    _person = null;
    setState(() {
      try {
        final String jsonText = _jsonTextController.text;
        debugPrint("JSON TEXT: $jsonText");
        var decoded = jsonDecode(jsonText);
        debugPrint("DECODED: type: ${decoded.runtimeType} value: $decoded");
        _person = Person.fromJson(decoded);
        debugPrint(
          "PERSON OBJECT: type: ${_person.runtimeType} value: " " $_person");
      } catch (e) {
        debugPrint("ERROR: $e");
        _error = e.toString();
      }
    });
  }
}

```

```

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text("Deserialization"),
        ),
        body: Center(
            child: Padding(
                child: ListView(
                    children: <Widget>[
                        _createJsonTextFormField(),
                        Padding(
                            padding: EdgeInsets.only(top: 10.0),
                            child: Text(
                                _error == null ? '' : 'An error occurred:\n\n$_error',
                                style: TextStyle(color: Colors.red))),
                        Padding(
                            padding: EdgeInsets.only(top: 10.0),
                            child: Text(
                                _person == null
                                    ? 'Person is null'
                                    : 'Converted to Person object:\n\n$_person',
                                )));
                    ],
                ),
                padding: EdgeInsets.all(10.0),
            ),
        ),
        floatingActionButton: FloatingActionButton(
            onPressed: _convertJsonToPerson,
            tooltip: 'Increment',
            child: Icon(Icons.refresh),
        ),
    );
}

```

8.3.2.2 Ecriture manuelle du code pour la sérialisation et la désérialisation

Cette approche utilise la classe `json` dans le package principal `dart:convert` pour effectuer une conversion entre les `maps` et les chaînes JSON.

- ⊕ Lors de la sérialisation d'un objet, nous écrivons du code pour convertir les données de notre classe en une `maps` afin que la classe `json` puisse ensuite la convertir en chaîne JSON.
- ⊕ Lors de la désérialisation d'une chaîne JSON, nous écrivons du code pour convertir la `maps` en données de notre classe.

La **première étape** est d'écrire une classe de données incluant les méthodes `toJson` et `fromJson`. Tout d'abord, vous devez écrire une classe de données Dart qui contiendra les données à sérialiser et contiendra les données une fois qu'elles auront été désérialisées.

- En cas de sérialisation : écrivez une méthode `toJson` qui renvoie une `maps` à partir des données de cette classe (voir la classe `Person` par exemple).
- En cas de désérialisation : écrivez une méthode de type `factory fromJson` qui crée une instance de la classe de données à partir d'un seul argument de type `maps`.

La **deuxième étape** est d'ajouter du code pour appeler la sérialisation/désérialisation de la classe de données.

```
class Person {
    final String name;
    final String addressLine1;
    final String addressCity;
    final String addressState;
    final List<Person> children;

    Person(this.name, this.addressLine1, this.addressCity, this.addressState,
          this.children);

    Map<String, dynamic> toJson() {
        var map = {
            'name': name,
            'addr': addressLine1,
            'city': addressCity,
            'state': addressState,
            'children': children
        };
        return map;
    }

    factory Person.fromJson(Map<String, dynamic> json) {
        if (json == null) {
            throw FormatException("Null JSON.");
        }
        List<dynamic> decodedChildren = json['children'];
        List<Person> children = [];
        decodedChildren.forEach((decodedChild) {
            children.add(Person.fromJson(decodedChild));
        });
        return Person(
            json['name'], json['addr'], json['city'], json['state'], children);
    }

    @override
    String toString() {
        return 'Person{name: $name, addressLine1: $addressLine1, addressCity: $addressCity, addressState: $addressState, children: $children}';
    }
}
```

En cas de sérialisation : invoquez `json.encode` dans la classe `json` dans le package principal `dart:convert`. La classe `json` appelle la méthode `toJson` dans votre classe de données pour créer une `maps`. La classe `json` convertit ensuite la `maps` en une chaîne JSON.

En cas de désérialisation : appelez `json[]` dans la classe `json` du package principal `dart:convert` pour renvoyer un élément d'une `maps`. La classe `json` convertira la chaîne JSON en une `maps`. Appelez la méthode de type `factory` `.fromJson` dans la classe de données pour convertir la `maps` en une instance de la classe de données.

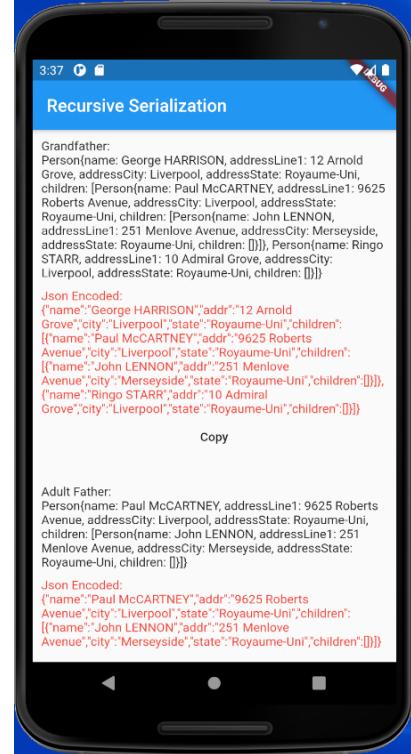
L'exemple qui est représenté par les applications « `serialize_manually` » et « `deserialize_manually` ». Ces deux applications illustrent quelque chose de plus complexe : la sérialisation / désérialisation manuelle récursive.

Dans ce cas le code généré ne fonctionne pas. De ce fait, nous sommes obligés d'utiliser l'autre méthode. Nous démontrons la sérialisation et la désérialisation d'un objet `Person` de manière récursive.

Ces objets `Person` peuvent avoir des enfants, qui à leur tour peuvent avoir des enfants, etc. Dans cet exemple, nous pouvons avoir des enfants et des petits-enfants.

L'application « `serialize_manually` » crée des objets `Person` pour toutes les personnes de la famille et les affiche via la méthode `toString()` (en noir).

Elle désérialise également chacun d'eux, affichant le JSON en dessous (en rouge). Il existe un bouton `Copy` pour copier le JSON dans le presse-papiers afin de pouvoir le coller dans un formateur JSON en ligne.



```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'person.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Serialize JSON',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
```

```

class MyHomePage extends StatelessWidget {
    static Person _grandchild = Person(
        "John LENNON", "251 Menlove Avenue", "Merseyside", "Royaume-Uni", []
    );
    static Person _adultFather = Person("Paul McCARTNEY", "9625 Roberts Avenue",
        "Liverpool", "Royaume-Uni", [_grandchild]);
    static Person _adultNoChildren =
        Person("Ringo STARR", "10 Admiral Grove", "Liverpool", "Royaume-
    Uni", []);
    static Person _grandfather = Person("George HARRISON", "12 Arnold Grove",
        "Liverpool", "Royaume-Uni", [_adultFather, _adultNoChildren]);

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(
                title: Text("Recursive Serialization"),
            ),
            body: Center(
                child: Padding(
                    child: ListView(
                        children: <Widget>[
                            Padding(
                                child: Text("Grandfather:\n$_grandfather"),
                                padding: EdgeInsets.only(top: 0.0),
                            ),
                            Padding(
                                child: Text(
                                    "Json Encoded:\n${json.encode(_grandfather)}",
                                    style: TextStyle(color: Colors.red),
                                ),
                                padding: EdgeInsets.only(top: 10.0),
                            ),
                            FlatButton(
                                child: Text("Copy"),
                                onPressed: () {
                                    Clipboard.setData(
                                        ClipboardData(text: "${json.encode(_grandfather)}"));
                                },
                            ),
                            Padding(
                                child: Text("Adult Father:\n$_adultFather"),
                                padding: EdgeInsets.only(top: 30.0),
                            ),
                            Padding(
                                child: Text(
                                    "Json Encoded:\n${json.encode(_adultFather)}",
                                    style: TextStyle(color: Colors.red),
                                ),
                                padding: EdgeInsets.only(top: 10.0),
                            ),
                        ],
                    ),
                ),
            ),
        );
    }
}

```

```

),
FlatButton(
    child: Text("Copy"),
    onPressed: () {
        Clipboard.setData(
            ClipboardData(text: "${json.encode(_adultFather)}"));
    },
),
Padding(
    child: Text("Adult No Children:\n$_adultNoChildren"),
    padding: EdgeInsets.only(top: 30.0),
),
Padding(
    child: Text(
        "Json Encoded:\n${json.encode(_adultNoChildren)}",
        style: TextStyle(color: Colors.red),
    ),
    padding: EdgeInsets.only(top: 10.0),
),
FlatButton(
    child: Text("Copy"),
    onPressed: () {
        Clipboard.setData(ClipboardData(
            text: "${json.encode(_adultNoChildren)}"));
    }),
Padding(
    child: Text("Grandchild:\n$_grandchild"),
    padding: EdgeInsets.only(top: 30.0)),
Padding(
    child: Text(
        "Json Encoded:\n${json.encode(_grandchild)}",
        style: TextStyle(color: Colors.red),
    ),
    padding: EdgeInsets.only(top: 10.0),
),
FlatButton(
    child: Text("Copy"),
    onPressed: () {
        Clipboard.setData(
            ClipboardData(text: "${json.encode(_grandchild)}"));
    }),
],
),
padding: EdgeInsets.all(10.0),
),
);
}
}

```

L'application « *deserialize_manually* » vous permet de saisir le JSON d'une personne, puis d'appuyer sur le bouton pour le déserialiser. En cas de succès, une méthode *toString()* de l'objet Person s'affiche en dessous (en noir). Si une erreur se produit (peut-être que vous avez entré un mauvais JSON?), elle est affichée en dessous (en rouge). N'oubliez pas que cela doit fonctionner de manière récursive - le JSON Person peut avoir des enfants, ce qui créera un objet Person avec des enfants (et ainsi de suite).

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

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

class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return MaterialApp(
            title: 'Deserialize JSON',
            theme: ThemeData(
                primarySwatch: Colors.blue,
            ),
            home: MyHomePage(),
        );
    }
}

class MyHomePage extends StatefulWidget {
    MyHomePage({Key key}) : super(key: key);

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

class _MyHomePageState extends State<MyHomePage> {
    final _jsonTextController = TextEditingController();
    Person _person;
    String _error;

    _MyHomePageState() {
        final String grandchild =
            "{\"name\":\"George HARRISON\", \"addr1\":\"12 Arnold Grove\", \"city\":
            \"Liverpool\", \"state\":\"Royaume-Uni\", \"children\":[\" + "
            "\"]}";
        final String adultFather =
            "{\"name\":\"Paul McCARTNEY\", \"addr1\":\"9625 Roberts Avenue\", \"city\"
            \":Liverpool\", \"state\":\"Royaume-Uni\", \"children\":[\" + "
            "\"]}";
    }
}
```

```
4:34
Recursive Deserialization
Enter the JSON for a person.
McCARTNEY", "addr1": "9625 Roberts Avenue", "city": "Liverpool", "state": "Royaume-Uni", "children": [{"name": "George HARRISON", "addr1": "12 Arnold Grove", "city": "Liverpool", "state": "Royaume-Uni", "children": []}], {"name": "Ringo STARR", "addr1": "10 Admiral Grove", "city": "Liverpool", "state": "Royaume-Uni", "children": []}]
Converted to Person object:
Person{name: John LENNON, addressLine1: null, addressCity: Merseyside, addressState: Royaume-Uni, children: [Person{name: Paul McCARTNEY, addressLine1: null, addressCity: Liverpool, addressState: Royaume-Uni, children: [Person{name: George HARRISON, addressLine1: null, addressCity: Liverpool, addressState: Royaume-Uni, children: []}], Person{name: Ringo STARR, addressLine1: null, addressCity: Liverpool, addressState: Royaume-Uni, children: []}]}}
```

```

        grandchild +
    "]}";
final String adultNoChildren =
"{"name\":\"Ringo STARR\", \"addr1\":\"10 Admiral Grove\", \"city\":\"Liverpool\",
\"state\":\"Royaume-Uni\", \"children\":[" +
    "]}";
final String grandfather =
"{"name\":\"John LENNON\", \"addr1\":\"251 Menlove Avenue\", \"city\":\"Merseyside\",
\"state\":\"Royaume-Uni\", \"children\":[" +
    adultFather +
    "," +
    adultNoChildren +
    "]}";
_jsonTextController.text = grandfather;
}

TextFormField _createJsonTextField() {
return new TextFormField(
validator: (value) {
if (value.isEmpty) {
return 'Please enter the json.';
}
},
decoration: InputDecoration(
border: OutlineInputBorder(),
hintText: 'Json',
labelText: 'Enter the json for a person.'),
controller: _jsonTextController,
autofocus: true,
maxLines: 8,
keyboardType: TextInputType.multiline,
);
}

_convertJsonToPerson() {
_error = null;
_person = null;
setState(() {
try {
final String jsonText = _jsonTextController.text;
debugPrint("JSON TEXT: $jsonText");
var decoded = json.decode(jsonText);
debugPrint("DECODED: type: ${decoded.runtimeType} value: $decoded");
_person = Person.fromJson(decoded);
debugPrint(
"PERSON OBJECT: type: ${_person.runtimeType} value: " "$_person");
} catch (e) {
debugPrint("ERROR: $e");
_error = e.toString();
}
});
}

```

```

        }
    });
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
            title: Text("Recursive Deserialization"),
        ),
        body: Center(
            child: Padding(
                child: ListView(
                    children: <Widget>[
                        _createJsonTextFormField(),
                        Padding(
                            padding: EdgeInsets.only(top: 10.0),
                            child: Text(
                                _error == null ? '' : 'An error occurred:\n\n$_error',
                                style: TextStyle(color: Colors.red))),
                        Padding(
                            padding: EdgeInsets.only(top: 10.0),
                            child: Text(_person == null
                                ? 'Person is null'
                                : 'Converted to Person object:\n\n$_person'))
                    ],
                ),
                padding: EdgeInsets.all(10.0),
            ),
        ),
        floatingActionButton: FloatingActionButton(
            onPressed: _convertJsonToPerson,
            tooltip: 'Increment',
            child: Icon(Icons.refresh),
        ),
    );
}
}

```

Cette application attribue par défaut votre entrée JSON initiale au grand-parent afin que vous puissiez voir cette récursivité fonctionner. Cette application écrit également sur la console afin que vous puissiez suivre ce qui se passe.