

Développement d'applications cross-plateformes avec Flutter

Formateur : Mehdi M'tir

Mehdi.mtir@gmail.com

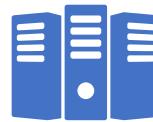




Formateur : Mehdi M'tir

- Développeur et chef de projets web depuis 2005
- Enseignant universitaire depuis 2006, spécialisé en développement web et mobile
- Formateur professionnel et consultant en technologies web et mobiles depuis 2013

Plan du module



- Partie 1  Introduction : Développement cross-plateforme et framework Flutter
- Partie 2  Présentation du langage Dart et du framework Flutter
- Partie 3  Les widgets : stateful vs stateless, arborescence et héritage, navigation
- Partie 4  Persistance des données : en local ou à distance, gestion de l'état
- Partie 5  Préparation à la mise en production : déploiement, test et sécurité

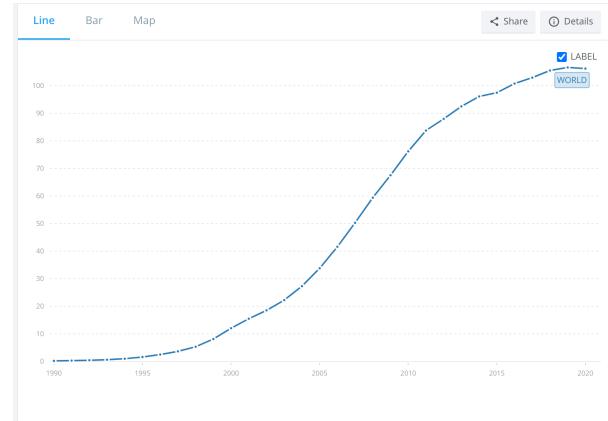
Partie 1

Introduction au développement des applications cross-plateformes

- Alternatives de développement mobile
- Cross-plateforme vs Hybride vs Natif
- Présentation du framework Flutter
- Préparation de l'environnement de travail
- Création d'une première application
- Architecture globale d'une application Flutter

Introduction

- D'abord considéré comme un simple support de communication, le mobile est devenu en quelques années un canal d'accès privilégié au web, contenus et services.
- A ce titre, les enjeux du mobile ont considérablement augmenté et font maintenant partie intégrante des stratégies des entreprises.
- Si l'importance du mobile est un fait globalement partagé, le sujet reste complexe à aborder pour la plupart des entreprises car il demeure nouveau, mouvant et à la croisée du web, du marketing et de la technique.



Mobile cellular subscriptions (per 100 people)

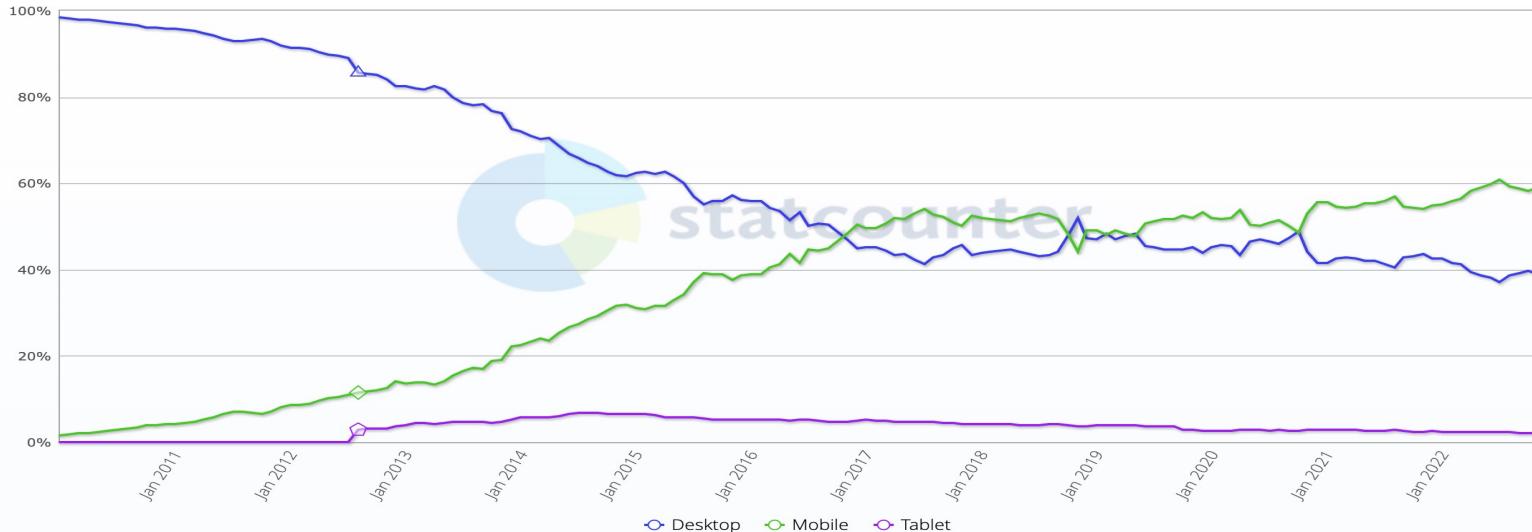
Source : <http://data.worldbank.org/indicator/IT.CEL.SETS.P2/countries?display=graph>

Chiffres clés

Desktop vs Mobile vs Tablet Market Share Worldwide

Jan 2010 - Nov 2022

Edit Chart Data



[Save Chart Image \(.png\)](#)

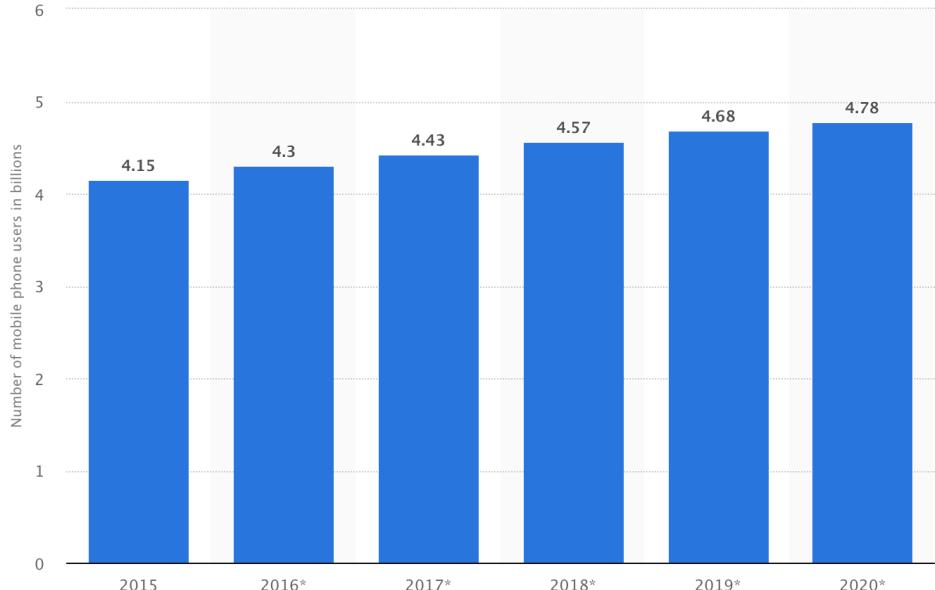
[Download Data \(.csv\)](#)

[Embed HTML](#)

<div id="desktop+mobile+tablet-comparison-ww-monthly-201001">

<https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-201001-202211>

Chiffres clés

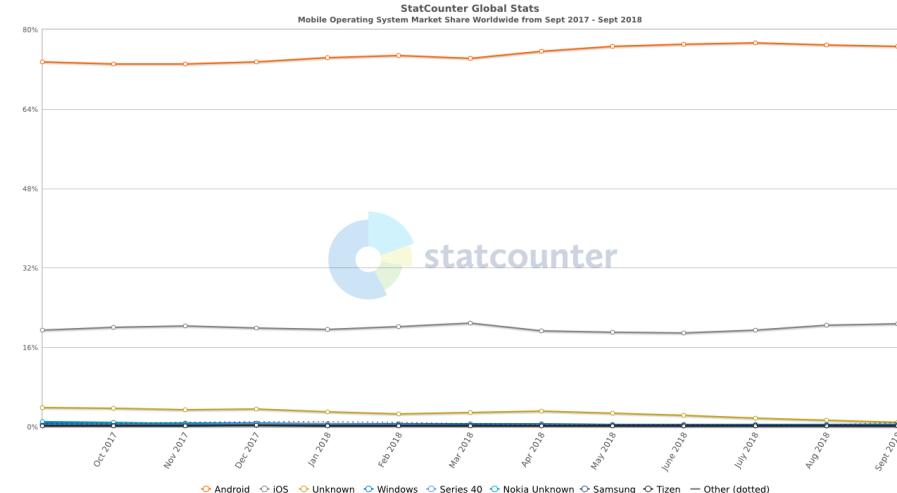
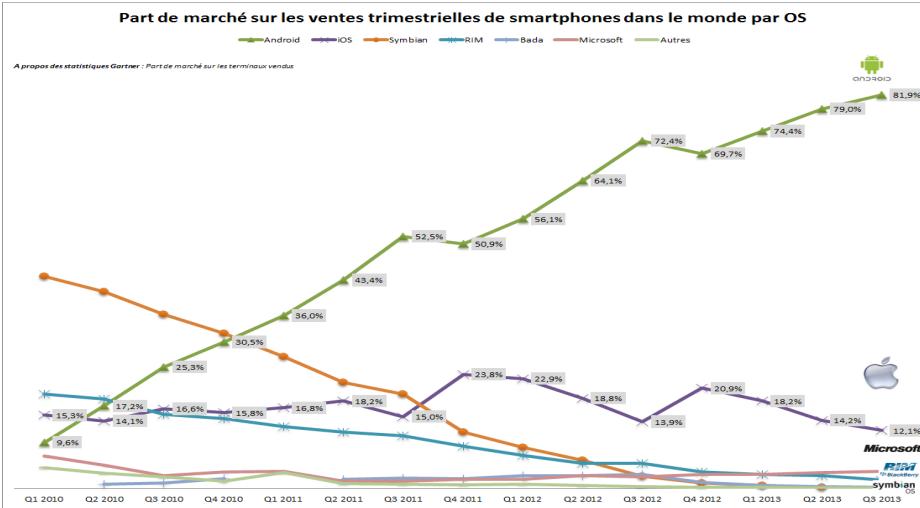


Source : <https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/>

"All people in the world are going to get a smartphone, and for most of them it will be their first computer."

**(Larry Page, CEO Google
2012)**

OS Mobiles



Source : <http://gs.statcounter.com/os-market-share/mobile/worldwide>

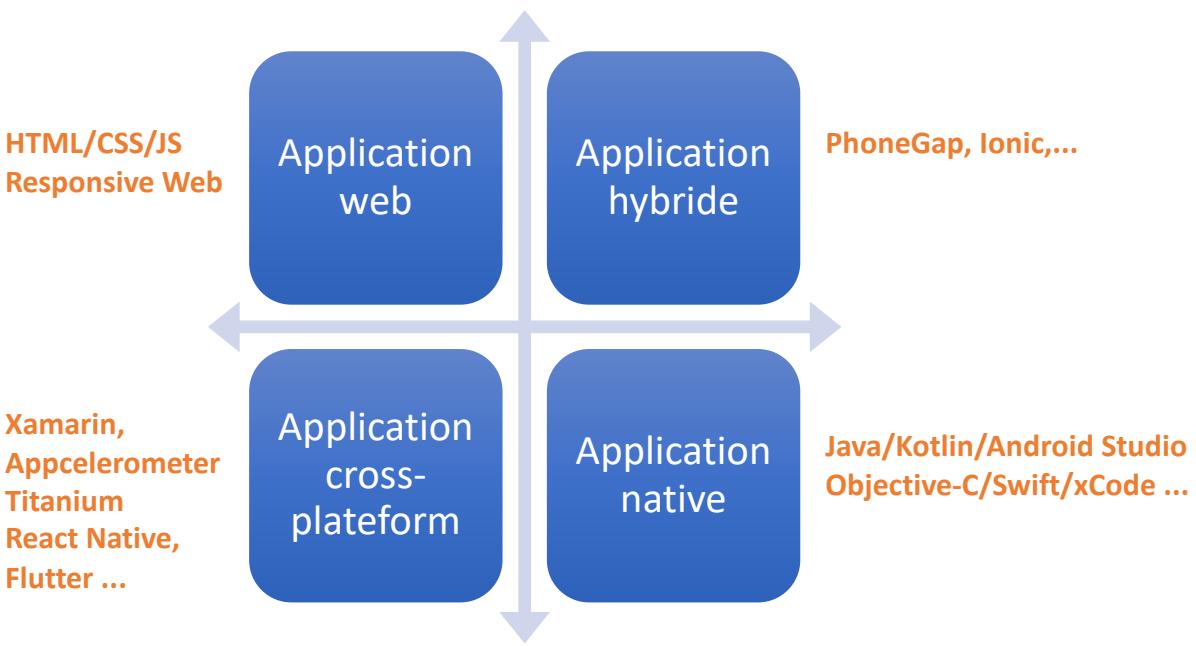
Développement d'applications mobiles

Les alternatives offertes aux développeurs



NATIF	HYBRIDE NATIF (cross-platform)	HYBRIDE WEB	WEB
<ul style="list-style-type: none">- Disponible sur stores- Accès direct aux ressources natives- Performances +++- Coût élevé- UI/UX native.- Apprentissage lent.	<ul style="list-style-type: none">- Disponible sur stores- Accès indirect aux ressources natives- Performances ++- Coût raisonnable- UI/UX native- Apprentissage +/- lent.	<ul style="list-style-type: none">- Disponibles sur stores.- Accès indirect aux ressources natives- Performances ++- Coût raisonnable- UI/UX web.- Apprentissage rapide	<ul style="list-style-type: none">- Accessible par URL.- Accès limité aux ressources natives- Performances ++- Coût bas.- UI/UX web- Apprentissage rapide

Alternatives pour le développement mobile



The Difference Between NATIVE, WEB & HYBRID MOBILE APPLICATIONS



Native applications are coded in the native language of the device (e.g Objective C for iOS, Java for Android). They are run directly on the device.

- ✓ Access Native APIs
- ✓ Distribute through App Stores
- ✗ Run on multiple platforms



Web applications are coded in HTML, CSS and JavaScript. They are served through the Internet and run through a browser.

- ✗ Access Native APIs
- ✗ Distribute through App Stores
- ✓ Run on multiple platforms



Hybrid applications are coded in HTML, CSS and JavaScript*. They are run through an invisible browser that is packaged into a native application.

- ✓ Access Native APIs
- ✓ Distribute through App Stores
- ✓ Run on multiple platforms

*There are other ways to create hybrid applications, but this is the most popular

- Android studio
- Java / Kotlin
- XML



androidcentral

- xCode
- Swift / Obj-C
- XML



Solutions natives

iOS



Principales solutions
hybrides natives



Principales solutions
hybrides web

Cross-plateformes vs Natif

- Aux débuts du mobile, pour développer une application, vous deviez utiliser l'ensemble des outils natifs de chaque plate-forme (Android, iOS...).
- Conséquences :
 - Construire en parallèle pour chaque plateforme mobile.
 - Gestion de plusieurs codes source.
 - Recrutement et rétention de développeurs natifs hautement spécialisés et coûteux.
- Pendant ce temps, la demande d'expériences mobiles continuait de croître de façon exponentielle.



Demand for mobile experiences is growing 5x faster than internal IT teams can deliver.

GAP

Cross-plateformes vs Natif

- Compte tenu du temps et du coût du développement natif traditionnel, de nombreuses équipes de développement avaient du mal à répondre à cette demande.
- ➔ Des solutions hybrides et/ou cross-plateformes commencent à apparaître.

Cross-plateformes vs Natif

Les principales raisons de passer du natif au cross-plateformes (ou hybride):

Rapidité du développement

- Construire pour plusieurs plates-formes à partir d'un seul code source permet souvent de fournir des applications multiplateformes 2 à 3 fois plus rapidement que le natif.

Efficacité

- La réduction des temps de développement et les coûts évités de recrutement et de rétention des talents natifs spécialisés peuvent permettre aux équipes d'économiser 60% ou plus.

UI / UX

- Avec un même code source exécuté sur le desktop, le mobile et le web, les applications cross-plateformes offrent une meilleure conception et une meilleure cohérence UX sur tous les canaux.

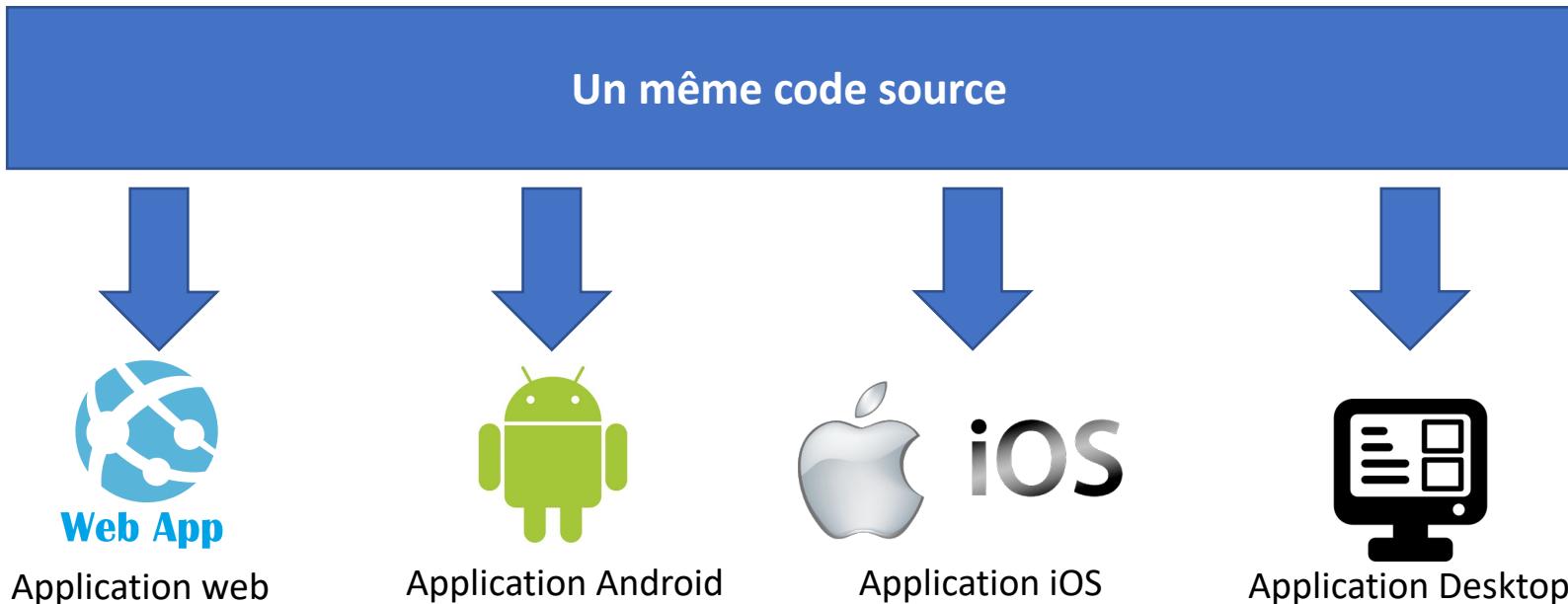
Compétences

- Le développement cross-plateforms donne aux développeurs web et aux entreprises disposant d'équipes Web internes les outils nécessaires pour créer des applications mobiles puissantes en utilisant leurs compétences et leurs talents existants.

Présentation du framework Flutter

- ❑ Flutter est un framework créé par **Google**.
- ❑ Framework cross-platforms pour le développement d'applications :
 - ❑ **Mobiles (Android et iOS)**
 - ❑ **Web,**
 - ❑ **Desktop (Windows, MacOS, Linux)**
- ❑ Basé sur le langage de programmation Dart (créé par **Google**).

Flutter en bref



Pourquoi utiliser Flutter?

Open source

Flutter est open Source.
Tout développeur peut l'utiliser dans n'importe quel but et sans frais.
Code source accessible sur :
<https://github.com/flutter/flutter>



open source
initiative

Cycle de développement rapide

Flutter est si rapide qu'il faut moins de 30 secondes pour la première compilation complète. Livré avec Hot-Restart & Hot-Restart.



Formation Flutter - Mehdi M'tir

Super productif

Grâce à la fonctionnalité de recharge à chaud (Hot – Reload) du widget Stateful, Flutter a un style de codage itératif très rapide.



Pourquoi utiliser Flutter?

Facilité d'apprentissage et partage de code

Toute personne ayant des connaissances de base en POO et en conception d'interface utilisateur peut facilement apprendre Flutter.



Bibliothèques de widgets

Flutter propose de nombreux widgets prêts à l'emploi que vous pouvez utiliser pour créer une application Flutter. Tels que : http, share plus, ...



Soutien communautaire

La communauté Flutter est plus petite que celle des frameworks web comme React. Mais Flutter se développe très rapidement.



Composants de Flutter

Un SDK (Software Development Kit)



Un ensemble d'outils pour :

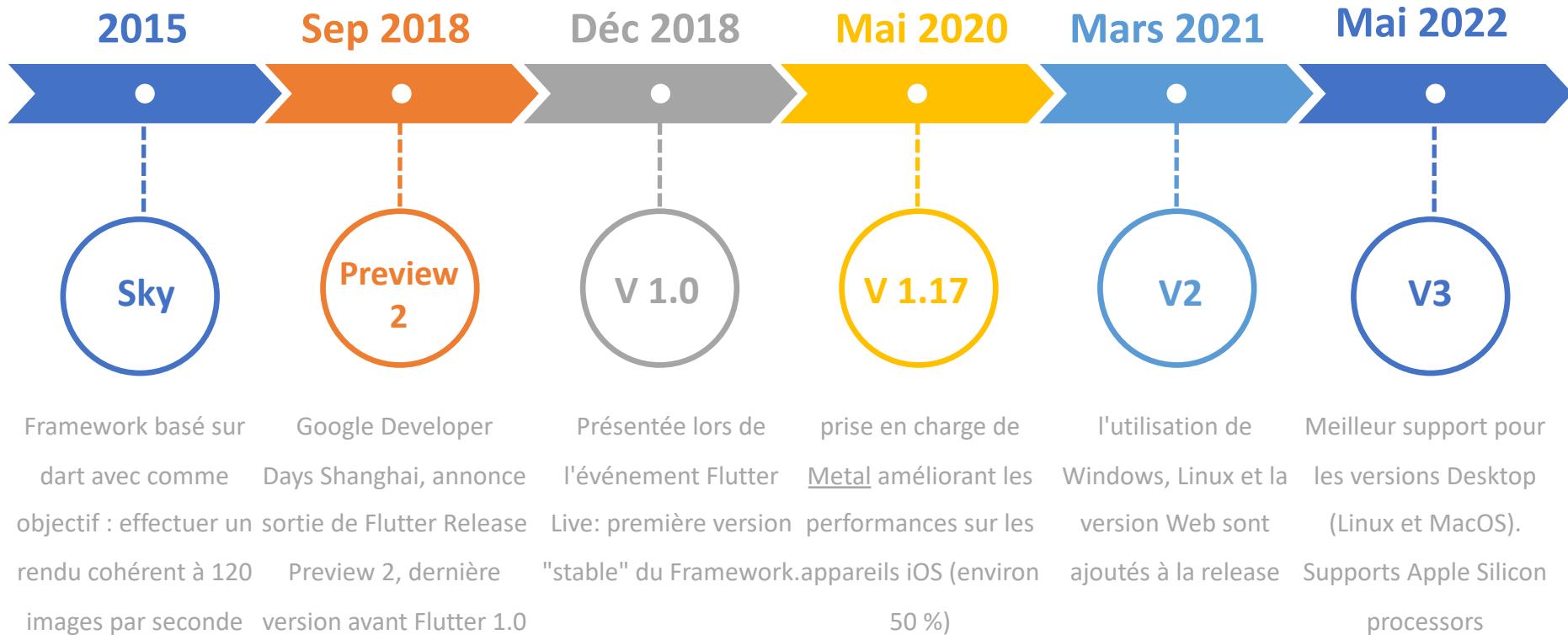
- compiler votre code source en code machine natif.
- faciliter les développements.

Une bibliothèque de Widgets



- Blocks de construction d'interface graphique réutilisables.
- Fonctions utilitaires.
- Packages.

Historique du framework Ionic



Préparation de l'environnement de travail

- Avant de commencer, nous avons besoin de :



Visual Studio Code



Il est recommandé
d'installer l'extension :
Flutter pour vsCode



<https://docs.flutter.dev/get-started/install>



Bien que non obligatoire, le système de contrôle de version Git est fortement recommandé.

Préparation de l'environnement de travail

Pour vérifier que l'environnement de travail est bien installé, vous pouvez lancer la commande :

```
flutter doctor
```

Un audit complet est fait avec des recommandations pour régler les problèmes rencontrés.

Créer une première application

- Après avoir correctement installé Flutter DDK et les EDIs (Visual Studio Code, Android Studio et xCode si vous avez un Mac), vous pouvez générer un nouveau projet à l'aide de votre EDI ou à partir d'une invite de commande en tapant :

flutter create first_app

- Un répertoire est généré avec tous les fichiers nécessaires pour créer une application simple de compteur numérique.
- Vous pouvez alors lancer VS Code et ouvrir le dossier du projet.
- Avant de lancer l'application, il est nécessaire de lancer un émulateur à partir du gestionnaire AVD d'Android Studio ou xCode si vous êtes sur Mac.
- Une fois votre Émulateur lancé, vous pouvez lancer l'application à partir du menu Exécuter de VS Code

Exécuter → Exécuter sans débogage

- Il est également possible de lancer l'application à partir de l'invite de commande

flutter run

```
> .dart_tool
> .idea
> android
> build
> ios
> lib
> linux
> macos
✓ test
> web
> windows
  .gitignore
  .metadata
  analysis_options....
  first_app.iml
  pubspec.lock
  pubspec.yaml
  README.md
```

Architecture globale d'une application Flutter

- Les dossiers android, ios, linux, macos, web et windows représentent les environnements cibles de notre application.
- Le répertoire lib/ contiendra tout le code source de notre application.
- Le fichier pubspec.yaml représente le fichier de configuration principal
- Tous les autres fichiers sont utilisés par le framework Flutter en interne et ne seront que très rarement modifiés.

Partie 2

Introduction au langage de programmation dart

- Présentation de Dart
- Historique
- Concepts importants
- Mots clés
- Variables et types
- Fonctions
- Opérateurs
- Structures de contrôle...

Présentation de Dart

- Dart est un langage de programmation moderne de haut niveau à usage général, développé à l'origine par Google.
- C'est un nouveau langage de programmation qui est apparu en 2011, mais sa version stable est sortie en juin 2017.
- Dart n'est pas si populaire à cette époque, mais il gagne en popularité lorsqu'il est utilisé par le Flutter.
- Dart est un langage de programmation dynamique, basé sur les classes et orienté objet avec closure et portée lexicale.
- Syntaxiquement, il est assez similaire à Java, C et JavaScript. Si vous connaissez l'un de ces langages de programmation, vous pouvez facilement apprendre le langage de programmation Dart.

Présentation de Dart

- Dart est un langage de programmation open source utilisé pour développer des applications mobiles, des applications Web modernes, des applications de bureau et des applications pour l'Internet des objets.
- C'est un langage compilé qui prend en charge deux types de techniques de compilation.
 - **AOT (Ahead of Time)** - Il convertit le code Dart en code JavaScript optimisé à l'aide du compilateur dart2js et s'exécute sur tous les navigateurs Web modernes. Il compile le code au moment de la construction.
 - **JIT (Just-In-Time)** - Il convertit le byte code en code machine (code natif), mais uniquement le code nécessaire.

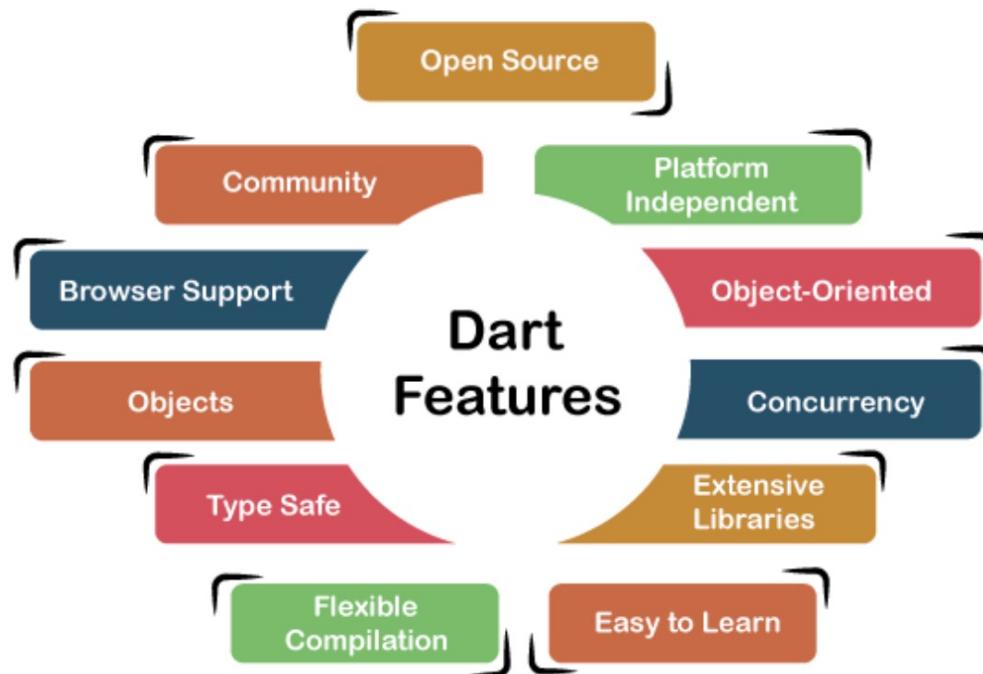
Historique de Dart

- Dart a été dévoilé pour la première fois lors de la conférence GOTO du 10 au 12 octobre 2011 à Aarhus, au Danemark. Il est initialement conçu par Lars Bark et Kespar et développé par Google.
- La première version 1.0 de Dart a été publiée le 14 novembre 2013, destinée à remplacer JavaScript
- En juillet 2014, la première édition du langage Dart a été approuvée par Ecma International lors de sa 107e Assemblée générale.
- La première version a été critiquée en raison d'un dysfonctionnement sur le Web et ce plan a été abandonné en 2015 avec la version 1.9 de Dart.
- La deuxième version de Dart 2.0 est sortie en août 2018, incluant un système de typage. La version la plus récente est 2.18.6 sortie en Décembre 2022.

Un programme simple avec dart

```
// Define a function.  
void printInteger(int aNumber) {  
    print('The number is $aNumber.');// Print to console.  
}  
  
// This is where the app starts executing.  
void main() {  
    var number = 42; // Declare and initialize a variable.  
    printInteger(number); // Call a function.  
}
```

Fonctionnalités de Dart



Concepts importants (1)

- Tout ce que vous pouvez placer dans une variable est un **objet**, et chaque objet est une instance d'une classe. Même les nombres, les fonctions et null sont des objets. À l'exception de null (si vous activez l'option sound null safety), tous les objets héritent de la classe **Object**.
- Bien que Dart soit fortement typé, les annotations de type sont facultatives car Dart peut déduire les types. Dans le code du slide précédent, la variable **number** est supposé être de type **int**.

Concepts importants (2)

- Si vous activez l'option **null safety**, les variables ne peuvent pas contenir null sauf si vous dites qu'elles le peuvent. Vous pouvez rendre une variable nullable en mettant un point d'interrogation (?) à la fin de son type. Par exemple, une variable de type **int?** peut être un entier ou nul.
- Si vous savez qu'une expression n'est jamais évaluée à null mais que Dart n'est pas d'accord, vous pouvez ajouter ! pour affirmer qu'il n'est pas nul (et lever une exception si c'est le cas). Un exemple : **int x = nullableButNotNullInt !**

Concepts importants (3)

- Lorsque vous voulez indiquer explicitement que n'importe quel type est autorisé, utilisez le type `Object?` (si vous avez activé null safety), `Object` ou, si vous devez différer la vérification de type jusqu'à l'exécution, le `type spécial dynamic`.
- Dart prend en charge les types génériques, comme `List<int>`(une liste d'entiers) ou `List<Object>`(une liste d'objets de n'importe quel type).

Concepts importants (4)

- Dart prend en charge les fonctions de niveau supérieur (telles que `main()`), ainsi que les fonctions liées à une classe ou à un objet (méthodes *statiques* et d' *instance* , respectivement). Vous pouvez également créer des fonctions dans des fonctions (fonctions *imbriquées* ou *locales*).
- De même, Dart prend en charge les variables de niveau supérieur , ainsi que les variables liées à une classe ou à un objet (variables statiques et d'instance). Les variables d'instance sont parfois appelées champs ou propriétés.

Concepts importants (5)

- Contrairement à Java, Dart n'a pas les mots-clés **public**, **protected** et **private**. Si un identifiant commence par un trait de soulignement (`_`), il est privé à sa bibliothèque.
- *Les identificateurs* peuvent commencer par une lettre ou un trait de soulignement (`_`), suivis de n'importe quelle combinaison de caractères plus des chiffres.
- Dart a à la fois *des expressions* (qui ont des valeurs d'exécution) et *des instructions* (qui n'en ont pas). Par exemple, l'[expression conditionnelle](#) `condition ? expr1 : expr2` a une valeur `expr1` ou `expr2`. Comparez cela à une **instruction if-else**

Mots clés (1)

- Le tableau suivant répertorie les mots que le langage Dart traite spécialement.

abstract ²	else	import ²	show ¹
as ²	enum	in	static ²
assert	export ²	interface ²	super
async ¹	extends	is	switch
await ³	extension ²	late ²	sync ¹
break	external ↗ ²	library ²	this
case	factory ²	mixin ²	throw

Mots clés (2)

catch

false

new

true

class

final

null

try

const

finally

on¹

typedef²

continue

for

operator²

var

covariant²

Function²

part²

void

default

get²

required²

while

deferred²

hide¹

rethrow

with

do

if

return

yield³

dynamic²

implements²

set²

Mots clés (3)

- Évitez d'utiliser ces mots comme identifiants. Cependant, si nécessaire, les mots-clés marqués d'exposants peuvent être des identifiants :
 - Les mots avec ¹ sont des mots-clés contextuels, qui n'ont de sens qu'à des endroits spécifiques. Ce sont des identifiants valides partout.
 - Les mots avec ² sont des identificateurs intégrés. Ces mots clés sont des identifiants valides dans la plupart des endroits, mais ils ne peuvent pas être utilisés comme noms de classe ou de type, ou comme préfixes d'importation.
 - Les mots avec ³ sont des mots réservés limités liés à la prise en charge asynchrone. Vous ne pouvez pas utiliser await ou yield comme identifiant dans un corps de fonction marqué par async, async* ou sync*.
- Tous les autres mots du tableau sont des mots réservés, qui ne peuvent pas être des identificateurs.

Les variables

- Voici un exemple de création d'une variable et de son initialisation :

```
var name = 'Bob';
```

- Les variables stockent les références. La variable appelée **name** contient une référence à un objet **String** avec une valeur de "Bob".
- Le type de la variable **name** est supposé être **String**. Si un objet n'est pas limité à un seul type, spécifiez le type **Object** (ou **dynamic** si nécessaire).

```
String name = 'Bob';
```

```
Object name = 'Bob';
```

Types prédéfinis

Le langage Dart a un support spécial pour les éléments suivants :

- [Numbers](#) (int, double)
- [Strings](#) (String)
- [Booleans](#) (bool)
- [Lists](#) (List)
- [Sets](#) (Set)
- [Maps](#) (Map)
- [Runes](#) (Runes; Souvent remplacés par l'API characters)
- [Symbols](#) (Symbol)
- La valeur null (Null)

Fonctions

Dart est un véritable langage orienté objet, donc même les fonctions sont des objets et ont un type [Function](#).

Cela signifie que les fonctions peuvent être affectées à des variables ou transmises en tant qu'arguments à d'autres fonctions.

Vous pouvez également appeler une instance d'une classe Dart comme s'il s'agissait d'une fonction.

```
bool isNoble(int atomicNumber) {  
    return _nobleGases[atomicNumber] != null;  
}
```

```
isNoble(atomicNumber) {  
    return _nobleGases[atomicNumber] != null;  
}
```

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

Opérateurs

Dart prend en charge les opérateurs indiqués dans le tableau suivant. Le tableau montre l'associativité et la priorité des opérateurs de Dart du plus élevé au plus bas, qui sont une **approximation** des relations d'opérateurs de Dart.

La description	Opérateur	Associativité
suffixe unary	<code>expr++ expr-- () [] ?[] . ?. !</code>	Aucun
préfixe unary	<code>-expr !expr ~expr ++expr --expr await expr</code>	Aucun
multiplicatif	<code>* / % ~/</code>	La gauche
additif	<code>+ -</code>	La gauche
décalage	<code><< >> >>></code>	La gauche
ET au niveau du bit	<code>&</code>	La gauche
XOR au niveau du bit	<code>^</code>	La gauche
OU au niveau du bit	<code> </code>	La gauche
test relationnel et de type	<code>>= > <= < as is is!</code>	Aucun
égalité	<code>== !=</code>	Aucun
ET logique	<code>&&</code>	La gauche
OU logique	<code> </code>	La gauche
si nul	<code>??</code>	La gauche
conditionnel	<code>expr1 ? expr2 : expr3</code>	Droite
Cascade	<code>... ?...</code>	Droite
mission	<code>= *= /= += -= &= ^= etc.</code>	Droite

Structures conditionnelles

```
if (isRaining()) {  
    you.bringRainCoat();  
} else if (isSnowing()) {  
    you.wearJacket();  
} else {  
    car.putTopDown();  
}
```

```
var visibility = isPublic ? 'public' : 'private';
```

```
var command = 'OPEN';  
switch (command) {  
    case 'CLOSED':  
        executeClosed();  
        break;  
    case 'PENDING':  
        executePending();  
        break;  
    case 'APPROVED':  
        executeApproved();  
        break;  
    case 'DENIED':  
        executeDenied();  
        break;  
    case 'OPEN':  
        executeOpen();  
        break;  
    default:  
        executeUnknown();  
}
```

Structures itératives

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

```
while (!isDone()) {
  doSomething();
}
```

```
var collection = [1, 2, 3];
collection.forEach(print); // 1 2 3
```

```
do {
  printLine();
} while (!atEndOfPage());
```

Classes

```
class Point {  
    double? x; // Declare instance variable x, initially null.  
    double? y; // Declare y, initially null.  
    double z = 0; // Declare z, initially 0.  
}
```

```
var p = Point(2, 2);  
  
// Get the value of y.  
assert(p.y == 2);  
  
// Invoke distanceTo() on p.  
double distance = p.distanceTo(Point(4, 4));
```

```
// If p is non-null, set a variable equal to its y value.  
var a = p?.y;
```

```
var p1 = Point(2, 2);  
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

====

```
var p1 = new Point(2, 2);  
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

Partie 3

Concepts de base du framework Flutter

- Architecture
- Widgets
- Layout
- Ajout de l'interactivité
- Navigation entre les pages
- Animation

Architecture de Flutter

- L'architecture Flutter est composée de quatre composants principaux.
 - Flutter Engine
 - Foundation Library
 - Core Widgets
 - Design Specific Widgets

Flutter Engine

- Le Flutter Engine est un environnement d'exécution portable pour les applications mobiles de haute qualité principalement basées sur le langage de programmation C++.
- Il comprend des animations et des graphiques, des E/S de fichiers et de réseau, une architecture de plug-ins, une prise en charge de l'accessibilité et un environnement d'exécution de Dart pour écrire, construire et exécuter des applications Flutter.
- Le package graphique open source de Google, Skia, est utilisé pour produire des visuels de bas niveau.

Foundation Library

- La bibliothèque Foundation fournit tous les packages nécessaires pour les éléments de base de la construction d'une application Flutter.
- Ces bibliothèques sont écrites dans le langage de programmation Dart.

Présentation des Widgets

- Dans une application Flutter tout est basé sur la notion de Widget : c'est la notion principale du framework.
- Un widget est un composant d'interface utilisateur (UI) qui impacte et contrôle l'affichage et l'interface de l'application.
- Il s'agit d'une description immutable d'une partie de l'interface utilisateur qui contient des graphiques, du texte, des formes et des animations.
- Les widgets Flutter sont construits à l'aide d'un framework moderne qui s'inspire de React .

Présentation des Widgets

- L'idée centrale est que vous créez votre interface utilisateur à partir de widgets.
- Il existe deux types de widgets : **stateless** et **stateful**
- Les widgets décrivent à quoi leur vue devrait ressembler compte tenu de leur configuration et de leur état actuels.
- Lorsque l'état d'un widget change, le widget reconstruit sa description, que le framework compare à la description précédente afin de déterminer les changements minimaux nécessaires dans l'arborescence de rendu sous-jacente pour passer d'un état à l'autre.

Application Flutter minimale

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

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Core Widgets (1)

- Flutter est livré avec une suite de puissants widgets de base.
- Ces Widgets peuvent être classés en deux catégories :
 - **Visibles** (Input et Output)
 - **Invisible** (Layout et Contrôle)
- Parmi les Widgets visibles nous pouvons citer :
- **Text**
 - Le Text widget vous permet de créer une série de texte stylé dans votre application.

Core Widgets (2)

- Button
 - Ce widget nous permet d'effectuer une action spécifique en un seul clic. Flutter ne prend pas en charge le widget Button de manière native ; à la place, il utilise des boutons comme le FlatButton et le RaisedButton.
- Image
 - Ce widget contient une image qui peut être récupérée à partir de diverses sources, y compris le dossier de ressources (assets) ou directement à partir de l'URL. Il a de nombreux constructeurs pour charger des images, qui sont listés ci-dessous :
 - **image** : Il s'agit d'un chargeur d'image générique utilisé par ImageProvider.
 - **asset** : Il charge les images du dossier d'asset de votre projet.
 - **file** : Il lit les images du dossier du système.
 - **memory** : charge l'image depuis la mémoire.
 - **network** : Il récupère les photos du réseau.

Core Widgets (3)

Parmi les widgets invisibles :

- Row, Column

- Ces widgets flexibles vous permettent de créer des mises en page flexibles dans les directions horizontale (Row) et verticale (Column). La conception de ces objets est basée sur le modèle de mise en page flexbox du Web.

- Center

- Ce widget est utilisé pour centrer le widget enfant qu'il contient.

- Padding

- Ce widget couvre d'autres widgets pour fournir un espace interne dans des directions spécifiques. L'espace peut également être fourni dans toutes les directions.

Core Widgets (4)

- Stack

- Au lieu d'être orienté linéairement (horizontalement ou verticalement), un Stackwidget vous permet de placer des widgets les uns sur les autres dans l'ordre de la peinture. Vous pouvez ensuite utiliser le [Positioned](#)widget sur les enfants de a Stackpour les positionner par rapport au bord supérieur, droit, inférieur ou gauche de la pile. Les piles sont basées sur le modèle de disposition de positionnement absolu du Web.

- Container

- Le Container widget vous permet de créer un élément visuel rectangulaire. Un conteneur peut être décoré avec un [BoxDecoration](#), tel qu'un arrière-plan, une bordure ou une ombre. A Containerpeut également avoir des marges, un rembourrage et des contraintes appliquées à sa taille. De plus, a Containerpeut être transformé dans un espace tridimensionnel à l'aide d'une matrice.

Core Widgets (5)

- Gestures
 - C'est un widget dans Flutter qui offre une interactivité (comment écouter et répondre aux gestes) en utilisant la classe GestureDetector. GestureDetector est un widget invisible qui interagit avec son widget enfant en appuyant, en faisant glisser et en mettant à l'échelle. La combinaison avec le widget GestureDetector peut également ajouter plus d'éléments interactifs dans les widgets actuels.
- State Management
 - Le widget StatefulWidget dans Flutter est utilisé pour conserver l'état du widget. Lorsque son état interne change, il est toujours restitué. Le nouveau rendu est optimisé en mesurant la distance entre l'ancienne et la nouvelle interface utilisateur du widget et en rendant les modifications nécessaires.

Core Widgets (6)

- Layers

- Les couches (layers) sont une notion clé dans le framework Flutter, classées en plusieurs catégories basées sur la complexité et organisées de manière descendante.
- La couche la plus élevée est l'interface utilisateur de l'application, unique aux plates-formes Android et iOS.
- Tous les widgets natifs Flutter sont situés sur la deuxième couche la plus élevée.
- La couche de rendu vient ensuite, responsable du rendu de tout dans l'application Flutter. Les couches appartiennent ensuite à Gestures, à la bibliothèque Foundation, au moteur et au code spécifique à la plate-forme principale.

Specific Widgets Design

Le framework Flutter a deux familles de widgets qui adhèrent à différents langages de conception.

- **Material Design** pour les applications Android.
- **Cupertino Style** pour les applications iOS.

Utiliser les widgets de Material Design

Flutter utilise également d'autres widgets fournis par des bibliothèques tierces tel que **Material Design**

Pour utiliser les widgets fournis par Material Design il faut d'abord s'assurer d'avoir une entrée `uses-material-design: true` dans la section **flutter** du fichier de configuration `pubspec.yaml`

```
name: my_app
flutter:
    uses-material-design: true
```

Au début des fichiers qui vont utiliser ce type de widget, il faut importer la bibliothèque Material

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

Material Design

Flutter utilise également d'autres widgets fournis par des bibliothèques tierces tel que **Material Design**

Pour utiliser les widgets fournis par Material Design il faut d'abord s'assurer d'avoir une entrée **uses-material-design: true** dans la section **flutter** du fichier de configuration **pubspec.yaml**

```
name: my_app
flutter:
    uses-material-design: true
```

Au début des fichiers qui vont utiliser ce type de widget, il faut importer la bibliothèque Material

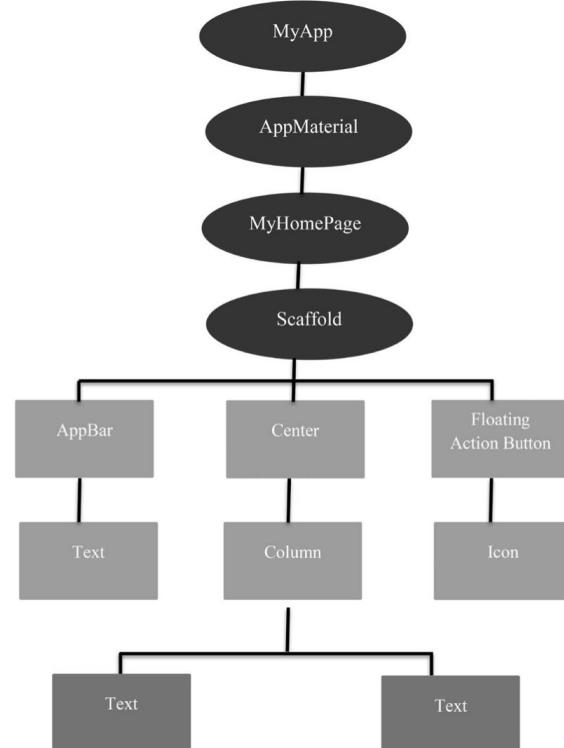
```
import 'package:flutter/material.dart';
```

Arbre de Widgets

Pour construire une application Flutter, les widgets sont imbriqués les uns dans les autres.

Cela signifie que la racine de votre application est un widget et que tout ce qui se trouve en dessous est également un widget.

Un widget, par exemple, peut afficher des éléments, définir la conception, gérer l'interaction, etc.



Création d'un Widget simple

Pour créer un nouveau Widget, il suffit de créer une classe qui hérite de StatelessWidget ou StatefulWidget

Exemple :

```
Class ImageWidget extends StatelessWidget {  
    // Class-Stuff  
}
```

Application Flutter avec plusieurs Widgets (1)

Tout le contenu de cette application sera défini dans le fichier :
lib/main.dart

Nous commençons par importer la bibliothèque Material

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

Application Flutter avec plusieurs Widgets (2)

Créer un Widget pour un AppBar personnalisé

```
class MyAppBar extends StatelessWidget {  
  const MyAppBar({required this.title, super.key});  
  
  // Fields in a Widget subclass are always marked "final".  
  
  final Widget title;  
  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      height: 56.0, // in logical pixels  
      padding: const EdgeInsets.symmetric(horizontal: 8.0),  
      decoration: BoxDecoration(color: Colors.blue[500]),  
      // Row is a horizontal, linear layout.  
      child: Row(  
        children: [  
          const IconButton(  
            icon: Icon(Icons.menu),  
            tooltip: 'Navigation menu',  
            onPressed: null, // null disables the button  
          ),  
          // Expanded expands its child  
          // to fill the available space.  
          Expanded(  
            child: title,  
          ),  
          const IconButton(  
            icon: Icon(Icons.search),  
            tooltip: 'Search',  
            onPressed: null,  
          ),  
        ],  
      ),  
    );  
  }  
}  
Formation Flutter - Me }
```

Application Flutter avec plusieurs Widgets (3)

Créer le widget principal (scaffold)

```
class MyScaffold extends StatelessWidget {  
  const MyScaffold({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    // Material is a conceptual piece  
    // of paper on which the UI appears.  
    return Material(  
      // Column is a vertical, linear layout.  
      child: Column(  
        children: [  
          MyAppBar(  
            title: Text(  
              'Example title',  
              style: Theme.of(context) //  
                  .primaryTextTheme  
                  .headline6,  
            ),  
            ),  
          const Expanded(  
            child: Center(  
              child: Text('Hello, world!'),  
            ),  
            ),  
          ],  
        ),  
      );  
    }  
}
```

Application Flutter avec plusieurs Widgets (4)

Créer la fonction main qui servira au lancement de l'application

```
void main() {  
  runApp(  
    const MaterialApp(  
      title: 'My app', // used by the OS task switcher  
      home: SafeArea(  
        child: MyScaffold(),  
      ),  
    ),  
  );  
}
```

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

void main() {
  runApp(
    const MaterialApp(
      title: 'Flutter Tutorial',
      home: TutorialHome(),
    ),
  );
}
```

```
class TutorialHome extends StatelessWidget {
  const TutorialHome({super.key});

  @override
  Widget build(BuildContext context) {
    // Scaffold is a layout for
    // the major Material Components.
    return Scaffold(
      appBar: AppBar(
        leading: const IconButton(
          icon: Icon(Icons.menu),
          tooltip: 'Navigation menu',
          onPressed: null,
        ),
        title: const Text('Example title'),
        actions: const [
          IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
          ),
        ],
      ),
      body: const Center(
        child: Text('Hello, world!'),
      ),
      floatingActionButton: const FloatingActionButton(
        tooltip: 'Add', // used by assistive technologies
        onPressed: null,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Utilisation des composants Material

Création d'un Widget stateful

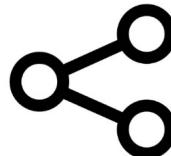
- Un StatefulWidget conserve des informations sur son état actuel.
- Il se compose principalement de deux classes : l'objet state et le widget.
- Il est dynamique car les données internes peuvent changer au cours de la durée de vie du widget.
- Ce widget n'a pas de fonction build().
- Il fournit une méthode appelée createState() qui renvoie une classe qui étend la classe State Flutter.
- Checkbox, Radio, Slider, InkWell, Form et TextField sont des instances de StatefulWidget.

Exemple d'un Widget stateful

```
class Cars extends StatefulWidget {  
    const Cars({ Key key, this.title }) : super(key:  
key);  
    @override  
    _CarState createState() => _CarState();  
}  
class _CarState extends State<Cars> {  
    @override  
    Widget build(BuildContext context) {  
        return Container(  
            color: const Color(0xFFEEFE),  
            child: Container(  
                child: Container( //child:  
Container() )  
            )  
        );  
    }  
}
```

Layouts (mises en pages)

- Le widget est la notion centrale du mécanisme de mise en page. Nous savons que Flutter traite tout comme s'il s'agissait d'un widget. Ainsi, l'image, l'icône, le contenu et même la mise en page de votre application sont tous des widgets.
- Certains des éléments que nous ne voyons pas sur l'interface utilisateur de notre application, tels que les lignes (Row), les colonnes (Column) et les grilles (Grid) qui organisent, contraignent et alignent les widgets visibles, sont également des widgets dans ce cas.
- Flutter nous permet de concevoir des widgets plus sophistiqués en assemblant de nombreux widgets. Par exemple, considérez l'image suivante, qui comporte trois symboles, chacun avec une étiquette en dessous.



share



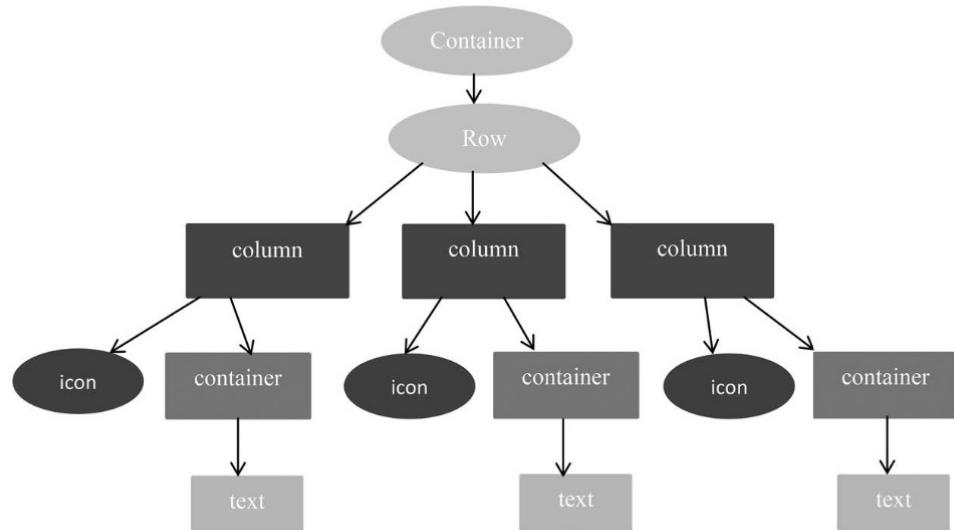
route



call

Layouts

- La disposition visuelle de l'image précédente peut être vue dans la figure suivante qui représente une rangée de trois colonnes, chacune avec une icône et une étiquette.



Un layout Flutter

Mettre en page un Widget

- Les instructions suivantes montrent comment mettre en page un widget :
- **Étape 1** : Tout d'abord, nous devons choisir un widget de mise en page.
- **Étape 2** : Après cela, créez un widget visible.
- **Étape 3** : Enfin, ajoutez le widget visible au widget de mise en page.
- **Étape 4** : Enfin, ajoutez le widget de mise en page à la page où vous souhaitez qu'il apparaisse.

Types de widgets de mise en page

- Il existe deux types de widgets de mise en page :
- **Widget à enfant unique (Single Child Widget)**
- **Widget à enfants multiples (Multiple Child Widget)**

Single Child Widget

- Un single child layout widget est un type de widget qui ne peut avoir qu'un seul widget enfant.
- Ces widgets peuvent également avoir une fonctionnalité de mise en page.
- Flutter nous offre une pléthore de Single Child Widgets pour améliorer l'interface utilisateur de l'application.
- Lorsque nous utilisons correctement ces widgets, nous gagnons du temps et rendons le code de l'application plus lisible.

Single Child Widget

- Voici des exemples de Single Child Widgets :
 - Container
 - Padding
 - Center
 - Align
 - SizedBox
 - AspectRatio
 - Baseline
 - ConstrainedBox
 - CustomSingleChildLayout
 - ...

Multiple Child Widget

- Les Multiple Child Widgets sont des widgets qui ont plus d'un widget enfant et leur layout est unique.
- Par exemple, un widget Row place ses widgets enfants horizontalement et un widget Column les placent verticalement.
- Lorsque nous combinons les widgets Row et Column, nous pouvons créer n'importe quel niveau complexité des widgets.

Multiple Child Widget

- Voici des exemples de Multiple Child Widgets :
 - Row
 - Column
 - Table
 - Flow
 - Stack
 - ...

Intéraction avec l'interface graphique

- Les gestion des évènements (Gestures) dans Flutter sont une fonctionnalité importante qui nous permet d'interagir avec l'application mobile (ou tout autre appareil tactile).
- En général, les gestes sont tout mouvement physique ou mouvement effectué par un utilisateur pour contrôler un appareil mobile.
- Flutter sépare le système gestuel en deux niveaux, qui sont décrits ci-dessous :
 1. Pointeurs
 2. Gestures

Pointeurs

- La première couche est constituée de pointeurs, qui représentent des données primaires concernant l'interaction de l'utilisateur.
- Elle fournit des événements qui expliquent l'emplacement et le mouvement des points sur les écrans, tels que les touches, les souris et le style. Flutter ne fournit pas de moyen d'annuler ou d'arrêter l'envoi d'événements de pointeur.
- Flutter a un widget Listener qui nous permet d'écouter les événements de pointeur directement à partir de la couche widgets.

Pointeurs

- Flutter a un widget Listener qui nous permet d'écouter les événements de pointeur directement à partir de la couche widgets.
- Les événements de pointeur sont classés en quatre types :
 - **PointerDownEvents** : cela permet au pointeur de toucher l'écran à une position particulière.
 - **PointerMoveEvents** : permet au pointeur de se déplacer d'un emplacement à l'écran à un autre.
 - **PointerUpEvents** : ces événements permettent au pointeur de quitter le contact avec l'écran.
 - **PointerCancelEvents** : cet événement est envoyé lorsque l'interaction du pointeur est terminée.

Gestures

- La deuxième couche représente les actions sémantiques telles que toucher, faire glisser et mettre à l'échelle détectées à partir de nombreux événements de pointeur individuels.
- Elle peut également envoyer plusieurs événements liés au cycle de vie des gestes, tels que le début du glissement, la mise à jour du glissement et la fin du glissement.

Gestures

- Voici quelques-uns des gestes les plus couramment utilisés :
 - Tap
 - Double Tap
 - Drag
 - Long Press
 - Pan
 - Pinch
 - ...

Gestion de l'état (State Management)

- Un état est une information qui peut être lue lors de la formation du widget et qui peut changer ou être ajustée tout au long de l'existence de l'application.
- Si nous souhaitons modifier notre widget, nous devons d'abord mettre à jour l'objet d'état, ce que nous pouvons accomplir en appelant la fonction `setState()` sur les widgets avec état.
- La fonction `set-State()` nous permet de modifier les attributs de l'objet d'état qui provoque le rafraîchissement de l'interface utilisateur.

Gestion de l'état (State Management)

- La gestion des états est l'une des activités les plus courantes et les plus essentielles du cycle de vie d'une application.
- Flutter, selon la documentation officielle, est déclaratif. Cela implique que l'interface utilisateur de Flutter est créée en reflétant l'état actuel de notre application

UI = f(state)

The layout on the screen Build methods The application state

Etat de l'application

- Supposons que nous ayons une liste de clients ou d'articles sur notre application.
- Supposons que nous ayons ajouté dynamiquement un nouveau client ou produit à cette liste.
- La liste doit ensuite être actualisée pour voir l'élément nouvellement ajouté dans l'enregistrement.
- Par conséquent, chaque fois que nous ajoutons un nouvel élément, nous devons recharger la liste.
- Ce type de programmation nécessite une gestion d'état pour gérer une telle circonstance et augmenter les performances.
- C'est parce que l'état est mis à jour chaque fois que nous apportons une modification ou que nous le mettons à jour.

Etat de l'application

- La gestion des états de Flutter est divisée en deux catégories conceptuelles, qui sont énumérées ci-dessous :
 - État éphémère.
 - État de l'application.

État éphémère

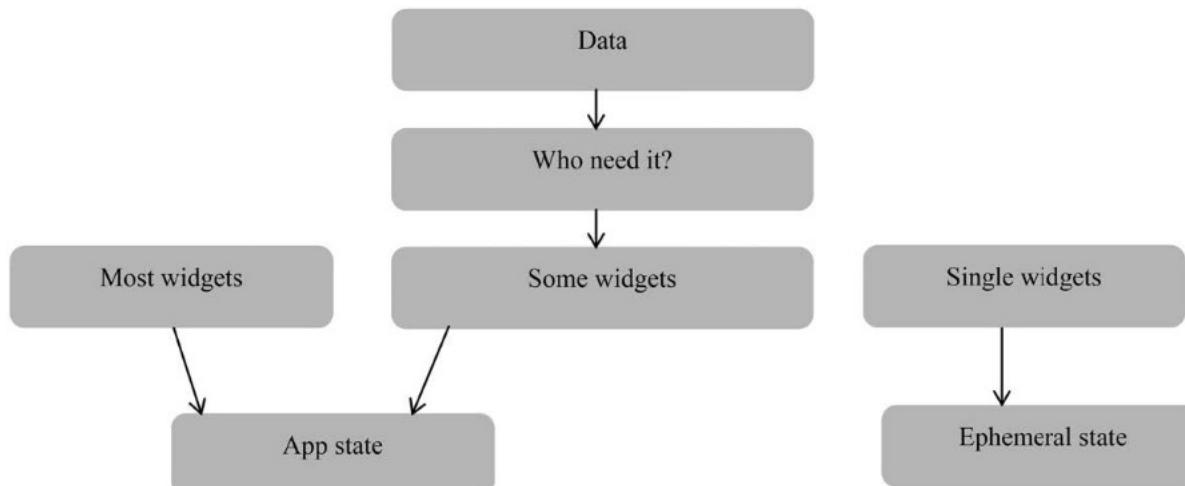
- L'état éphémère est également appelé état de l'interface utilisateur ou état local.
- C'est une sorte d'état associé à un widget particulier, ou on peut dire que c'est un état contenu dans un seul widget.
- Nous n'avons pas besoin d'utiliser des techniques de gestion de l'État dans ce type d'État.
- Le champ de texte est un exemple fréquent de cette condition.

État de l'application

- Ce n'est pas la même chose que l'état éphémère. C'est une forme d'état que nous souhaitons conserver tout au long des sessions utilisateur et répartir sur différentes parties de notre programme.
- Par conséquent, cette forme d'état peut être appliquée globalement.
- Il est parfois appelé état d'application ou état partagé.
- Les préférences de l'utilisateur, les informations de connexion, les alertes dans une application de réseau social, le panier d'achat dans une application de commerce électronique, le statut lu/non lu des articles dans une application d'actualités, etc. sont des exemples de cet état.

État éphémère vs État de l'application

- La figure suivante montre clairement la distinction entre l'état éphémère et l'état de l'application.



Navigation et routing

- Flutter fournit un système complet pour naviguer entre les écrans et gérer les liens profonds (deep links).
- Les petites applications sans liens profonds complexes peuvent utiliser **Navigator**, tandis que les applications avec des exigences spécifiques en matière de liens profonds et de navigation doivent utiliser le **Router** pour gérer correctement les liens profonds sur Android et iOS, et pour rester synchronisées avec la barre d'adresse lorsque l'application s'exécute sur le Web.

Utilisation du Navigator

- Le widget `Navigator` affiche les écrans sous forme de pile en utilisant les animations de transition selon la plate-forme cible.
- Pour naviguer vers un nouvel écran, accédez aux méthodes `Navigator` via la route `BuildContext` etappelez des méthodes impératives telles que `push()` or `pop()`:

```
onPressed: () {
  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (context) => const SongScreen(song: song),
    ),
  );
},
child: Text(song.name),
```

Utilisation des routes nommées

- Les applications nécessitant une navigation simple et des liens profonds peuvent utiliser `Navigator` pour la navigation et `MaterialApp.routes` pour les liens profonds :

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    routes: {
      '/': (context) => HomeScreen(),
      '/details': (context) => DetailScreen(),
    },
  );
}
```

Limites des routes nommées

- Bien que les routes nommées puissent gérer des liens profonds, le comportement est toujours le même et ne peut pas être personnalisé.
- Lorsqu'un nouveau lien profond est reçu par la plate-forme, Flutter en rajoute une nouvelle **Route** sur le **Navigator**, quel que soit l'endroit où se trouve actuellement l'utilisateur.
- Flutter ne prend pas non plus en charge le bouton Suivant du navigateur pour les applications utilisant des itinéraires nommés.
- Pour ces raisons, nous vous déconseillons d'utiliser des routes nommées dans la plupart des applications.

Utiliser le Router

- Les applications Flutter avec des exigences de navigation et de routage avancées (telles qu'une application Web qui utilise des liens directs vers chaque écran ou une application avec plusieurs `Navigator` widgets) doivent utiliser un package de routage tel que `go_router` qui peut analyser le chemin d'accès et configurer le `Navigator` chaque fois que l'application reçoit un nouveau lien profond.
- Pour utiliser le `Router`, passez au `Router constructor` sur `MaterialApp` ou `CupertinoApp` et fournissez-lui une `Router configuration`.
- Les packages de routage, tels que `go_router` , vous fournissent généralement une configuration.

Exemple avec Router

```
MaterialApp.router(  
    routerConfig: GoRouter(  
        // ...  
    )  
);
```

Partie 4

Persistante des données

- Solutions de sauvegarde des données en local.
- Persistante des données avec SQLite
- Communication avec un serveur distant

Solutions de sauvegarde des données en local

- Pour sauvegarder les données d'une application Flutter en local, 4 alternatives sont possibles :
- **Shared Preferences** : données stockées sous forme de paires clé/valeur.
- **Secure Storage** : paires clé/valeurs pour les données sensibles. Stockage plus sécurisé.
- **Hive / SQLite** : pour sauvegarder des données fortement structurées.
- **Local File Storage** : pour sauvegarder des données semi-structurées dans des fichiers. Plusieurs formats peuvent être choisis tels que : texte simple, csv, json ...

Persistance des données avec SQLite

- Si vous développez une application qui doit conserver et interroger de grandes quantités de données sur l'appareil local, envisagez d'utiliser une base de données au lieu d'un fichier local ou d'un magasin clé-valeur.
- En général, les bases de données fournissent des insertions, des mises à jour et des requêtes plus rapides par rapport aux autres solutions de persistance locales.
- Les applications Flutter peuvent utiliser les bases de données **SQLite** via le plugin **sqflite** disponible sur **pub.dev**.

Communication avec un serveur distant

- Le package http fournit le moyen le plus simple d'émettre des requêtes http. Ce package est pris en charge sur Android, iOS et le Web.
- Configuration des permissions sous Android :

```
<manifest xmlns:android...>
    ...
    <uses-permission android:name="android.permission.INTERNET" />
    <application ...
</manifest>
```

AndroidManifest.xml

Communication avec un serveur distant

- Configuration des permissions sous iOS :

```
<key>com.apple.security.network.client</key>
<true/>
```

Fichier .entitlements

Partie 5

Préparation à la mise en production

- Modes de build
- Persistance des données avec SQLite
- Communication avec un serveur distant

Modes de build sous Flutter

- Flutter prend en charge trois modes lors de la compilation de votre application et un mode pour les tests (headless mode). Vous choisissez un mode de compilation en fonction de votre position dans le cycle de développement.
 - Utilisez le mode **débogage (debug)** pendant le développement, lorsque vous souhaitez utiliser le rechargement à chaud .
 - Utilisez le mode **profil (profile)** lorsque vous souhaitez analyser les performances.
 - Utilisez le mode de **publication (release)** lorsque vous êtes prêt à publier votre application.

Debug mode

- En mode débogage , l'application est configurée pour le débogage sur l'appareil physique, l'émulateur ou le simulateur.
- Le mode débogage pour les applications mobiles signifie que :
 - Les assertions sont activées.
 - Les extensions de service sont activées.
 - La compilation est optimisée pour des cycles de développement et d'exécution rapides (mais pas pour la vitesse d'exécution, la taille binaire ou le déploiement).
 - Le débogage est activé et les outils prenant en charge le débogage au niveau de la source (tels que DevTools) peuvent se connecter au processus.
- Le mode débogage pour une application Web signifie que :
 - La construction n'est pas minifiée et l'arborescence n'a pas été effectuée.
 - L'application est compilée avec le compilateur dartdevc pour un débogage plus facile.

Release mode

- Utilisez le *mode de publication* pour déployer l'application, lorsque vous souhaitez une optimisation maximale et une taille d'empreinte minimale.
- Pour mobile, le mode release (qui n'est pas supporté sur le simulateur ou l'émulateur), signifie que :
 - Les assertions sont désactivées.
 - Les informations de débogage sont supprimées.
 - Le débogage est désactivé.
 - La compilation est optimisée pour un démarrage rapide, une exécution rapide et des paquets de petite taille.
 - Les extensions de service sont désactivées.
- Le mode de publication d'une application Web signifie que :
 - Le build est minifié et un tree shaking a été effectué.
 - L'application est compilée avec le compilateur [dart2js](#) pour de meilleures performances.

Profile mode

- En mode profil , une certaine capacité de débogage est conservée, suffisamment pour profiler les performances de votre application.
- Le mode profil est désactivé sur l'émulateur et le simulateur, car leur comportement n'est pas représentatif des performances réelles.
- Sur mobile, le mode profil est similaire au mode release, avec les différences suivantes :
 - Certaines extensions de service, telles que celle qui active la superposition de performances, sont activées.
 - Le traçage est activé et les outils prenant en charge le débogage au niveau de la source (tels que DevTools) peuvent se connecter au processus.
- Le mode profil pour une application Web signifie que :
 - Le build n'est pas minifié mais un tree shaking a été effectué.
 - L'application est compilée avec le compilateur dart2js .

Tester une application Flutter

- Plus votre application possède de fonctionnalités, plus il est difficile de la tester manuellement.
- Les tests automatisés permettent de s'assurer que votre application fonctionne correctement avant de la publier, tout en conservant la rapidité de votre fonctionnalité et de la résolution des bogues.
- Les tests automatisés se répartissent en plusieurs catégories :
 - Un **test unitaire** teste une seule fonction, méthode ou classe.
 - Un **test de widget** (dans d'autres UI frameworks appelé test de composant) teste un seul widget.
 - Un **test d'intégration** teste une application complète ou une grande partie d'une application.

Tester une application Flutter

- Plus votre application possède de fonctionnalités, plus il est difficile de la tester manuellement.
- Les tests automatisés permettent de s'assurer que votre application fonctionne correctement avant de la publier, tout en conservant la rapidité de votre fonctionnalité et de la résolution des bogues.
- Les tests automatisés se répartissent en plusieurs catégories :
 - Un **test unitaire** teste une seule fonction, méthode ou classe.
 - Un **test de widget** (dans d'autres UI frameworks appelé test de composant) teste un seul widget.
 - Un **test d'intégration** teste une application complète ou une grande partie d'une application.

Code Dart obscurcissant

- L' obscurcissement du code est le processus de modification du binaire d'une application pour le rendre plus difficile à comprendre pour les humains. L'obscurcissement masque les noms de fonction et de classe dans votre code Dart compilé, ce qui rend difficile pour un attaquant de procéder à l'ingénierie inverse de votre application propriétaire.

```
$ flutter build apk --obfuscate --split-debug-info=<project-name>/<directory>
```

Créer des saveurs (flavors) pour Flutter

- Les saveurs (appelées *configurations de construction* dans iOS) vous permettent (le développeur) de créer des environnements distincts pour votre application en utilisant la même base de code.
- Par exemple, vous pouvez avoir une version pour votre application de production à part entière, une autre en tant qu'application "gratuite" limitée, une autre pour tester des fonctionnalités expérimentales, etc.

Créer et publier une application Android

- Au cours d'un cycle de développement typique, vous testez une application en utilisant flutter run la ligne de commande ou en utilisant les options **Exécuter** et **Déboguer** dans votre IDE. Par défaut, Flutter crée une version de *débogage* de votre application.
- Lorsque vous êtes prêt à préparer une version finale de votre application, par exemple pour la publier sur le Google Play Store, vous pouvez suivre les étapes suivantes :
 - Ajout d'une icône de lanceur
 - Activation des composants de matériau
 - Signature de l'application
 - Réduire votre code avec R8
 - Activation de la prise en charge du multidex
 - Examen du manifeste de l'application
 - Examen de la configuration de construction
 - Construire l'application pour la publication
 - Publication sur le Google Play Store
 - Mise à jour du numéro de version de l'application

Créer et publier une application iOS

- Pré-requis :
- Xcode est nécessaire pour créer et publier votre application. Vous devez utiliser un appareil exécutant macOS pour suivre ce guide.
- Avant de commencer le processus de publication de votre application, assurez-vous qu'elle respecte les directives d'examen des applications d'Apple .
- Pour publier votre application sur l'App Store, vous devez d'abord vous inscrire au programme pour développeurs Apple . Vous pouvez en savoir plus sur les différentes options d'adhésion dans le guide Choisir un abonnement d'Apple.

Créer et publier une application iOS

- Pré-requis :
- Xcode est nécessaire pour créer et publier votre application. Vous devez utiliser un appareil exécutant macOS pour suivre ce guide.
- Avant de commencer le processus de publication de votre application, assurez-vous qu'elle respecte les directives d'examen des applications¹ d'Apple .
- Pour publier votre application sur l'App Store, vous devez d'abord vous inscrire au programme pour développeurs Apple² . Vous pouvez en savoir plus sur les différentes options d'adhésion dans le guide Choisir un abonnement d'Apple³.
- ¹ : <https://developer.apple.com/app-store/review/>
- ² : <https://developer.apple.com/programs/>
- ³ : <https://developer.apple.com/support/compare-memberships/>

Créer et publier une application iOS

- **Etapes :**
- Enregistrez votre application sur App Store Connect
 - Enregistrer un ID de forfait
 - Créer un enregistrement d'application sur App Store Connect
- Examiner les paramètres du projet Xcode
- Mettre à jour la version de déploiement de l'application
- Ajouter une icône d'application
- Ajouter une image de lancement
- Créer une archive de build et la télécharger sur App Store Connect
 - Mettre à jour les numéros de build et de version de l'application
 - Créer un ensemble d'applications
 - Télécharger le bundle d'applications sur App Store Connect

Merci pour votre attention