Découvrir les bases de développement multiplateforme

M216

Support du cours développement mobile

ETTAHERI Nizar
ettaheri.nizar@gmail.com


Dart

INTRODUCTION

Dart est un langage de programmation optimisé pour les applications sur plusieurs plate-formes. Il est développé par Google et est utilisé pour créer des applications mobiles, de bureau et web. Dart est un langage orienté objet, basé sur la classe. Il existe plusieurs environnements permettant de programmer dans le langage Dart tels que l'invite de commande, l'environnement web, un terminal smartphone.

LES VARIABLES EN DART

Les variables sont, en programmation, des conteneurs dans lesquels on sauvegarde des valeurs qui peuvent être de différents types (Ex: Entier, Réel). Ils sont utilisés dans le programme au cours de son exécution. La déclaration de variable en Dart se fait de la manière suivante :

```
type nomDeVariable;
```

NB : Nous mettrons l'accent, dans cet article, sur les syntaxes de base du code Flutter en ce qui concerne la déclaration des variables. Cependant depuis la version 2.0.0 de Flutter, la notion de null-safety voit le jour.

LES TYPE DES VARIABLES

Le langage Dart offre la possibilité d'utiliser plusieurs types pour typer nos variables. Dart est un langage fortement typé. Cependant, Dart intègre également le mode « soft type » qui permet au programmeur de ne pas donner de type précis à ses variable si il le souhaite. Mais cette dernière pratique est à utiliser avec précotions.

Les "Number"

Les number ou nombres en Dart peuvent être classés en 2 catégories.

- Les int : entier de taille arbitraire. Le type de données int est utilisé pour représenter des nombres entiers.
- Les double : nombres à virgule flottante 64 bits (double précision), comme spécifié par la norme IEEE 754. Le type de données double est utilisé pour représenter des nombres fractionnaires (Ex: 2.5).

Exemples :

```dart
int var_name;      // déclare une variable entière
double var_name;   // déclare une variable fractionnaire

//-----------------------------------------------------------------

void main() {
  // déclare une variable entière
  int num1 = 10;

  // déclare une variable fractionnaire
  double num2 = 10.50;

  // affichage des valeurs
  print(num1);
  print(num2);
}

//-----------------------------------------------------------------
// sortie dans la console
10
10.5
```

Les “String”

Le type de données String représente une séquence de caractères. Une chaîne Dart est une séquence d’unités de code UTF 16. Les valeurs de chaîne dans Dart peuvent être représentées à l’aide de guillemets simples, doubles ou triples. Les chaînes à une seule ligne sont représentées à l’aide de guillemets simples ou doubles. Les guillemets triples sont utilisés pour représenter des chaînes multi-lignes.

```dart
//guillemets simples
String  variable_name = 'value';

// Ou guillemets doubles
String  variable_name = "value";

// Ou guillemets triples
String  variable_name = '''ligne1
ligne2
ligne3''';

// Ou
String  variable_name= """ligne1
ligne2
ligne3""";
```

Les “Boolean”

Dart fournit un support intégré pour le type de données booléen. Le type de données booléen en DART ne prend en charge que deux valeurs : vrai ou faux. Le mot-clé bool est utilisé pour représenter un littéral booléen en DART. La syntaxe pour déclarer une variable booléenne dans DART est la suivante :

```
bool var_name1 = true;
// OU
bool var_name2 = false;
```

Les collections de donnée « List », « Map »

Les listes

Les listes en Dart, ou encore collection de données indicées, sont des variables dans lesquels vous pouvez stocker une succession de données comme dans un placard et accéder à ces données en précisant l'indice de la donnée à lire comme dans un tableau en javaScript ou PHP. Les listes sont des objets qui peuvent rétrécir ou s'agrandir au cours de l'exécution du programme, on dit qu'ils sont dynamiques.

```dart
//Pour déclarer un tableau vide
List<int> var_name = [];
//NB : 'int' étant facultatif

//Ou
var var_name = [1, 2, 3, 4];

//Ajout de l'élément 5 dans la liste var_name
var_name.add(5);

//Affichage du contenu de var_name
print(var_name); //sortie [1, 2, 3, 4, 5]
```

Plusieurs opérations peuvent être effectuées sur les listes on a :

- add() : pour ajouter un nouvel élément en fin de liste
- length : getter pour obtenir la taille de la liste
- reversed : getter renverser l'ordre des éléments de la liste
- remove( ) : pour supprimer un élément en précisant l'élément de la liste à supprimer. Etc.

Les “map”

L’objet map est une simple paire clé/valeur. Les clés et les valeurs d’une map peuvent être de n’importe quel type. Une map est une collection dynamique. En d’autres termes, la collection map peut croître et rétrécir tout comme les listes au moment de l’exécution.

```
Map<String, dynamic> var_name = { }; //Pour déclarer une map vide
//NB : ‘<String, dynamic>’ et ‘type_valeur’ étant facultatifs

var_name = {‘nom’: ‘John’, ‘prenom’: ‘Doe’};
var_name.addEntries( {‘age’: 12} );
var_name[“tel”] = “07xxxxxxx”;
print(var_name); //sortie {nom: John, prenom: Doe, age: 12, tel: 07xxxxxxxx}
```

Comme les listes les map peuvent être manipulées avec plusieurs fonctions :

- addAll() : Ajoute toutes les paires clé-valeur de other à cette map.
- clear( ) : Supprime toutes les paires de la map.
- remove( ) : Supprime la clé et sa valeur associée, si elle est présente, de la map. Etc.

LES OPERATEURS

Les opérateurs disponibles en Dart sont :

Opérateurs arithmétiques

Opérateurs relationnels

Opérateurs logiques

Opérateurs d’assignation

LES STRUCTURES CONDITIONNELLES

Instructions conditionnelles dans Dart sont classées dans le tableau suivant :

if

Une instruction if consiste en une expression booléenne suivie d'une ou plusieurs instructions.

```
void main(){
   if(a == b){
      print("vrai");
   }
}
```

If…else

Un if peut être suivi d'un bloc else facultatif. Le bloc else s'exécutera si l'expression booléenne testée par le bloc if est évaluée à false.

```
void main(){
   if(a == b){
      print("vrai");
   }else{
      print("faux");
   }
}
```

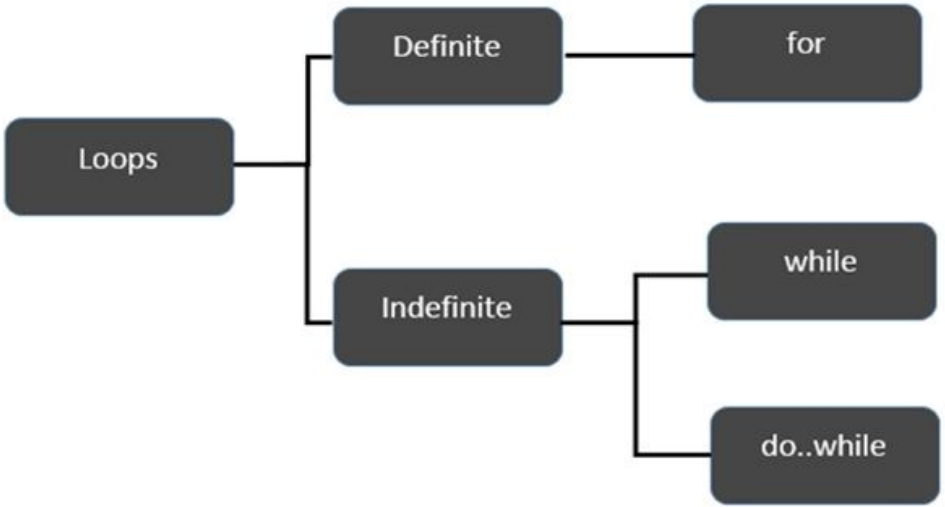
switch…case

L'instruction switch évalue une expression, fait correspondre la valeur de l'expression à une clause case et exécute les instructions associées à cette case.

```
void main(){
 int x = ....;
   switch(x){
      case 1 :
        print("1");
        break;
      case 2 :
         print("2");
        break;
      default :
        print("1");
   }
}
```

LES STRUCTURES ITERATIVES

Souvent, certaines instructions nécessitent une exécution répétée. Les boucles sont un moyen idéal de faire la même chose. Une boucle représente un ensemble d'instructions qui doivent être répétées. Dans le contexte d'une boucle, une répétition est appelée une itération. La figure suivante illustre la classification des boucles.



Boucles finies

- Boucle for

```
void main() {
  var obj = [12,13,14];
  for (int i =0; i<= obj.length; i++) {
      print(obj[i]);
  }
}
```

- Boucle for…in

```
void main() {
 var obj = [12,13,14];
 for (var prop in obj) {
     print(prop);
 }
}
```

Boucles infinies

- Boucle while

```
while(num >=1) {
 factorial = factorial * num;
 num--;
}
```

- Boucle do…while

```
do {
  print(n);
  n--;
} while(n>=0);
```

LES FONCTIONS EN DART

Déclaration et paramètres

Les fonctions sont des sous-programmes. Elles permettent de structurer le code en subdivisant le code en plusieurs parties qui réalisent des actions précises. Les fonctions en Dart sont de la forme :

```
//Déclaration de la fonction;
[type_de_retour] function_name (paramètres){
   //Corps de la fonction;
}
```

On distingue plusieurs type de paramètres :

- Paramètres positionnels optionnels

  Pour spécifier des paramètres de position facultatifs, utilisez des crochets [ ].

```
//Déclaration
void afficher([String var1, String var2]){...}

//Appel
afficher();
//ou
afficher("salut");
//ou
afficher("salut", "Les amis");
```

- Paramètres nommés optionnels

Contrairement aux paramètres positionnels, le nom du paramètre doit être spécifié lors de la transmission de la valeur. L'accolade {} peut être utilisée pour spécifier des paramètres nommés facultatifs.

```
  //Déclaration
void afficher({String var1, String var2}){...}

//Appel
afficher(var1: "salut");
//ou
afficher(var1: "salut", var2: "Les amis");
```

NB : en ajoutant le mot clé « required » devant le type d'un paramètre cette variable devient un paramètre nommé obligatoire.

```dart
//Déclaration
void afficher({required String var1, String var2}){...}

//Appel
afficher(var1: "salut");
//ou
afficher(var1: "salut", var2: "Les amis");

//Erreur
afficher();
//Ou
afficher(var2: "Les amis");
```

- **Compilations JIT (Just In Time) et AOT (Ahead Of Time)**

Dart est le principal langage de programmation pour développer des applications mobiles multiplateformes à l'aide du framework Flutter.

Dart utilise deux méthodes principales pour la compilation des programmes, la méthode **juste-à-temps** et la méthode anticipée **.** *Dans cet article, vous apprendrez ce que signifie chacune des méthodes de compilation et une comparaison entre les deux méthodes.*

Qu'est-ce que le juste à temps (JAT) ?

Un compilateur JIT convertit le code source du programme en code machine natif juste avant l'exécution du programme.Les compilateurs sont généralement l'un des facteurs déterminants de la vitesse d'une application, tant en développement qu'en phase de production. Les compilateurs JIT peuvent être utilisés pour améliorer la vitesse des performances et améliorer le temps d'exécution des applications. Il est courant dans le langage de programmation JAVA et constitue également l'une des façons dont les codes sources du programme sont exécutés dans DART. Les compilateurs juste-à-temps essaient de prédire quelles instructions seront exécutées ensuite afin de pouvoir compiler le code à l'avance.

Une façon courante de dire cela est qu'un compilateur JIT compile le code à la volée, ou en d'autres termes, juste à temps.

La machine virtuelle Dart dispose d'un compilateur juste-à-temps (JIT) qui prend en charge à la fois l'interprétation pure (comme requis sur les appareils iOS, par exemple) et l'optimisation de l'exécution. Cela est nécessaire pour un cycle de développement rapide, permettant de basculer facilement entre l'écriture des codes et les tests sur les appareils à la volée.

"Juste à temps" signifie fabriquer "uniquement ce qui est nécessaire, quand il le faut et dans la quantité nécessaire".

Qu'est-ce que l'Ahead-of-Time (AOT) ?

Le compilateur AOT fonctionne en compilant votre code avant qu'il ne soit "livré" à n'importe quel environnement d'exécution qui exécute le code. Le compilateur AOT est généralement utilisé lorsque l'application est prête à être déployée sur une boutique d'applications ou sur un backend de production interne.

Le code compilé par AOT s'exécute dans un runtime Dart efficace qui applique le système de type Dart sonore et gère la mémoire à l'aide d'une allocation d'objet rapide et d'un ramasse-miettes générationnel, comme spécifié sur le site Web de documentation de Dart.

Un instantané AOT pour une application de fléchettes peut être généré en ajoutant *-k aot*.

```
dart2native bin/main.dart -k aot
```

Voici ce que l'équipe officielle de Dart a à dire à ce sujet :Le code compilé à l'avance (AOT) avec un compilateur tel que dart2native présente des caractéristiques de performances différentes du code compilé juste-à-temps (JIT) dans la machine virtuelle Dart. Le code compilé par AOT est garanti pour avoir un démarrage rapide et des performances d'exécution cohérentes, sans latence lors des premières exécutions. Le code compilé par JIT est plus lent au démarrage, mais il peut avoir de meilleures performances de pointe après une exécution suffisamment longue pour que les optimisations d'exécution soient appliquées

La principale différence qui existe entre la compilation Ahead-Of-Time et Just-In-Time est l'heure à laquelle la compilation se produit.

- Conventions de codage, noms et ordonnancement

CamelCase

Pour déclarer une variable et des méthodes dans la programmation de fléchettes, nous utilisons l'approche camelCase. Dans camelCase, le premier mot de départ est mis en minuscule tandis que le mot suivant doit commencer par une majuscule. définissons quelques méthodes et variables en utilisant camelCase dans dart.

```
//variables
int testingNumbers = 1;
bool isItWorking = vrai;
```

PascalCase

Pour déclarer une classe et des énumérations dans Dart, nous utilisons l'approche PascalCase. En PascalCase, tous les mots commencent par une majuscule. comprenons avec quelques exemples en fléchettes.

```
// Classes dans
```

```
la classe de fléchettes StudentsRecord {}
```

```
classe Utilisateur test{}
```

serpent_case

Dans snack_case, nous avons des mots minuscules joints par une barre oblique descendante

_ . il n'est pas directement utilisé dans le code, il est utilisé pour nommer les fichiers.

comprenons-le avec quelques exemples.

```
//nom du fichier
```

```
main_file .dart
```

```
test_user_data .dart
```

STREAMING_CASE

il est spécifiquement utilisé pour les données constantes et non modifiables dans notre code

dans un cas où nous avons des valeurs constantes dans notre programme pour l'identifier facilement, nous utilisons STREAMING_CASE dans lequel toutes les lettres sont en majuscules et les mots sont joints avec la barre oblique descendante _ . prenons quelques exemples en fléchettes.

```
// valeurs constantes dans fléchette


const USER_ID = 1 ; constante TEST_DATA = 2 ;
```

C'est tout ce que vous devez savoir sur la convention de dénomination dans Dart, vous pouvez désormais nommer facilement vos programmes conformément aux règles de Dart pour la convention de dénomination.

- **Programmer orientée objet avec Dart**

Notion de POO en Dart

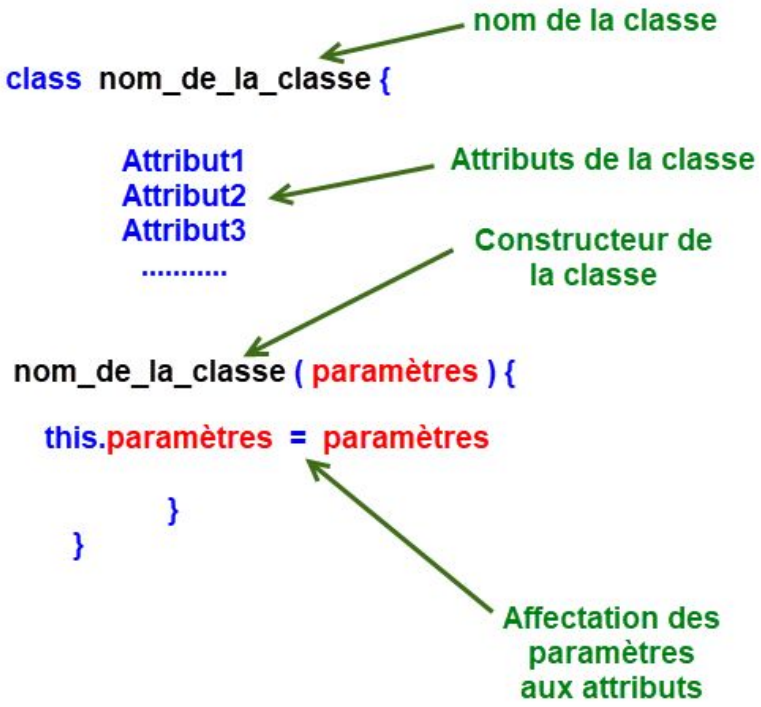
La programmation orientée objet, ou POO, est un paradigme de programmation qui permet de structurer les programmes de manière à ce que les propriétés et les comportements soient regroupés dans des objets à part. Par exemple, un objet peut représenter une personne avec un nom, un âge, une adresse, etc., avec des comportements tels que marcher, parler, respirer et courir.

En d’autres termes, la programmation orientée objet est une approche permettant de modéliser des éléments concrets du monde réel tels que les voitures les personnes..., ainsi que des relations entre des entités telles que les entreprises et les employés, les étudiants et les enseignants, etc. La modélisation POO modélise des entités réelles sous la forme d’objets logiciels certaines données qui leur sont associées et peuvent remplir certaines fonctions.

Structure générale d'une classe Dart

Une classe Dart possède exactement la même structure qu'une classe Java:

1. **Déclaration** à l'aide l'instruction **class**
2. **Déclaration des attributs** à l'aide des mots clé: **String, int, var, final**...
3. **Déclaration** du **constructeur** sous le **même nom** de la **classe.**
4. **Affectation des paramètres aux attributs** via l'instruction **this.**
5. **Instanciation** via la commande **new**


nom de la classe
class nom_de_la_classe {
Attribut1
Attribut2
Attribut3
.....
Attributs de la classe
Constructeur de la classe
nom_de_la_classe (paramètres) {
this.paramètres = paramètres
}
}
Affectation des paramètres aux attributs

Exemple :

```
class Personne {
  String name ;
  int age ;
// constructeur de la classe
  Personne(this.name , this.age);
}
void main() {
  var pers = new Personne("Aladin", 23);
  print("${pers.name} est agé de  ${pers.age} ans !");
  // Affiche: Aladin est agé de  27 ans !
}
```

Les méthodes de classe

Les méthodes de classe sont des fonctions déclarées au sein de la classe permettant de calculer ou effectuer des tâches spécifique. Exemple pour une classe rectangle qui a deux attributs: **longueur** et **largeur,** on peut ajouter une **méthode** qui calcul **la surface,** une **méthode** qui calcul le **périmètre ...**

```
class Rectangle {
  int longueur ;
  int largeur  ;

  Rectangle(this.longueur , this.largeur);
  // méthode qui calcul la surface du rectangle
  int surface(){
    return this.longueur * this.largeur;
  }
  // méthode qui calcul le périmètre du rectangle
  int perimetre(){
    return 2 * (this.longueur + this.largeur);
  }
  }
  void main() {
  var R = new Rectangle(10 , 5 );
  print("La surface du rectangle R est : ${R.surface()}  ");
  // Affiche: La surface du rectangle R est : 50
  print("Le périmètre du rectangle R est : ${R.perimetre()}  ");
  // Affiche: Le périmètre du rectangle R est : 30
}
```

L'héritage

Le langage supporte l'héritage, il n'y a pas d'héritage multiple. Pour qu'une classe hérite d'une autre, on utilise le mot clé **extends**. Reprenon l'exemple de notre classe Point, nous allons créer une classe Point3D pour ajouter "z" aux coordonnées du point.

```
Point(this.x, this.y);

Point(x, y) {
        this.x = x;
        this.y = y; }

 class Point3D extends Point {
        int z;

        Point3D(int x, int y, int this.z) : super(x,y);
 }

 main() {
        Point3D p3 = new Point3D(4, 5, 6);
        print("x=${p3.x} y=${p3.y} z=${p3.z}");
 }
```

interface

Il n'existe pas d'interface en tant que tel comme en Java. Pour utiliser ce concept, il faut passer par une classe abstraite. Chaque classe déclare une interface implicite du même nom, il est donc possible d'écrire : "class A implements Point { ... }"

La surchage d'opérateur

Tout comme le langage C++, Dart offre la possibilité de surcharger les opérateurs pour les instances d'une classe. Par exemple si l'on souhaite pouvoir additionner deux instances de notre classe point, nous ajoutons la méthode suivante :

```
Point operator +(Point p) {
    return new Point(x+p.x, y+p.y);
    }
```

L'addition de l'objet Point(2,3) et de Point(6,4) donne un nouveau point : Point(8,7). Voici les 19 opérateurs qui peuvent être surchargés :

<	+	|	[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
-	%	>>	

Erreur vs assertions vs exception – Quelle est la différence ?

Dans Dart, nous avons trois types d'erreurs différents qui peuvent se produire lors du développement et de l'exécution de votre application :

- **Les erreurs** dans Dart sont utilisées pour faire savoir à un consommateur d'une API ou d'une bibliothèque qu'il l'utilise mal. Cela signifie que les erreurs ne doivent pas être gérées et doivent planter votre application. C'est bien parce qu'ils ne se produisent que pendant le développement et informent le développeur que quelque chose va dans la mauvaise direction.
- **Les assertions** sont similaires aux erreurs et sont utilisées pour signaler des états incorrects qui ne devraient jamais se produire. La différence est que les assertions ne sont vérifiées qu'en mode débogage. Ils sont complètement ignorés en mode production.
- **Les exceptions** concernent un mauvais état attendu qui peut se produire lors de l'exécution. Étant donné que des exceptions sont attendues, vous devez les intercepter et les gérer de manière appropriée.

Pour attraper une erreur, nous devons envelopper la méthode qui jette l'erreur dans un `Try` bloque.

La `Try` bloc doit suivre un `catch` bloc qui a un paramètre d'exception de type objet.

Exemple :

```
try {
return api.getBird();
} catch (exception) {
   log(e)
}
```

Comme la plupart des langages de programmation, Dart nous offre également un moyen d'exécuter du code appelé qu'une erreur se soit produite ou non. Pour cela, nous pouvons utiliser le

`finally` bloquer. La `finally` block est par exemple souvent utilisé pour arrêter une sorte d'animation de chargement qui s'affichait pendant l'appel.

```
try {
return api.getBird();
} catch (exception) {
   log(e)
} finally {
  // Executed after the try block if no error occured
  // or after the catch block if an error occured
}
```

- **Les widgets de base**

Les widgets Flutter sont les principaux composants d’une application. Ils donnent l’apparence en fonction de leur configuration et de leur état.

Dans Flutter, tout ce qui apparaît dans l’interface utilisateur s’appelle widget et ils héritent tous de la classe **Widget**.

Lorsque vous créez une interface utilisateur dans Flutter, vous le faites en attachant chaque widget à l’écran de l’application afin qu’il s’adapte exactement à l’endroit où vous le souhaitez.

En fait les widgets Flutter, sont les boutons, les champs de texte, l'animations, les conteneurs, … et même une page plus grande. Tout ce qui apparaît à l'écran ou interagit avec celui-ci est un widget.

Des widgets partout ! En les imbriquant ensemble, vous créerez une hiérarchie appelée **Arborescence des Widgets**.

Une application est un widget de niveau supérieur et son interface utilisateur est construite à l'aide d'un ou plusieurs widgets enfants. Par conséquent, cette fonctionnalité de composants nous aide à créer des interfaces utilisateur plus complexes.

L'un des avantages de flutter est que nous pouvons éditer ces widgets, qui sont très flexibles à utiliser, et écrire nos propres widgets rapidement.

L’arborescence des widgets Flutter

Les widgets Flutter sont organisés dans une arborescence de parents-enfants, pour former ce que vous voyez à l’écran. Cette arborescence est tous les widgets que vous utilisez pour construire l’application, c’est-à-dire que le code que vous écrivez créera cette structure.

L’état des widgets Flutter

Chaque widgets Flutter peut être **sans état** ou **avec état**. La principale différence est la possibilité de restituer les widgets au moment de l’exécution. Le widget sans état ne sera utilisé qu’une seule fois et est immuable. Par contre un widget avec état peut être utilisé plusieurs fois en fonction du changement de son état interne.

StatefulWidget

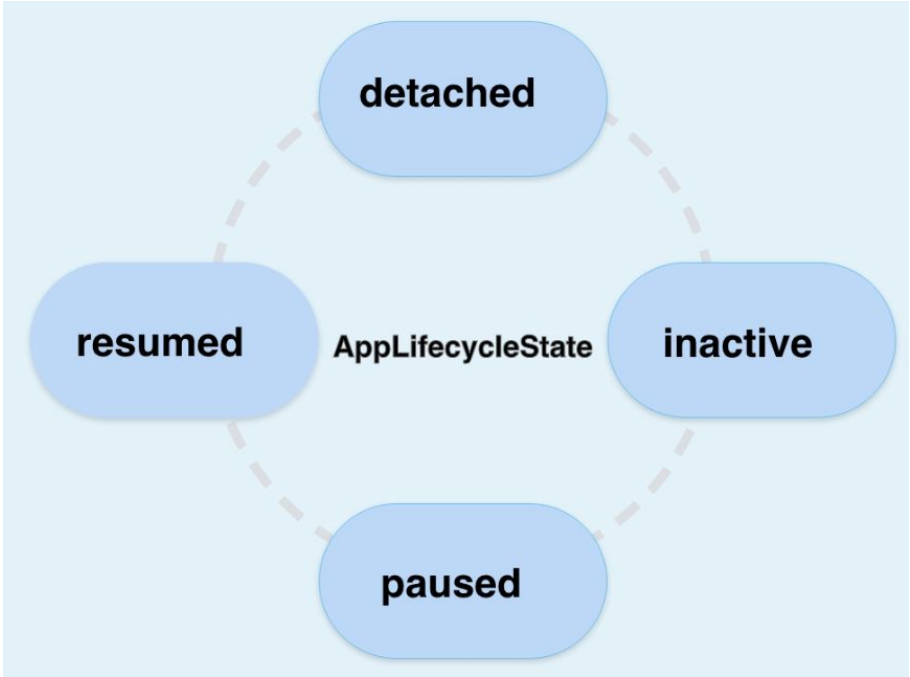
Un widget avec état est utilisé lorsqu'une partie doit changer de manière dynamique pendant l'exécution. En fait, il peut subir plusieurs modification pendant l'exécution de l'application. Donc, il peut suivre les modifications et mettre à jour l'interface utilisateur.

Les widgets avec état sont utiles lorsque la partie de l'interface utilisateur que nous décrivons change de manière dynamique. Si nous créons un bouton qui se met à jour chaque fois qu'un utilisateur clique dessus, il s'agit d'un widget avec état.

StatelessWidget

Un widget sans état ne peut pas changer pendant l'exécution d'une application Flutter. Cela signifie qu'il est impossible de le modifier pendant que l'application est en action. Pour cette raison, l'apparence et les propriétés restent inchangées pendant toute la durée de vie du widget.

De plus, il ne stocke pas les valeurs susceptibles de changer. Les widgets sans état peuvent être utiles lorsque la partie de l'interface utilisateur que nous décrivons ne dépend d'aucun autre widget. Par exemple, le texte, les icônes et les boutons d'icônes.


detached
resumed
AppLifecycleState
inactive
paused

Le cycle de vie des applications Flutter. Et nous verrons ensemble comment rendre disponible les méthodes du cycle de vie dans une application.

Lorsqu'un utilisateur navigue dans une application mobile, en sort et y revient, le système d'exploitation(Android ou iOS) envoie des notifications pour décrire son état. Ces différents états du cycle de vie sont définis par AppLifecycleState :

- **detached** : lorsque l'application est dans cet état, le moteur de Flutter est en cours d'exécution mais nous n'avons accès à aucune vue. Cela peut se produire lors de la première initialisation du moteur, soit après la destruction de la vue en raison de l'appel de la fonction pop() du Navigator.
- **inactive** : l'application est dans un état inactif et ne reçoit pas d'entrée utilisateur.
  Sur iOS, les applications passent à cet état lors d'un appel téléphonique, en réponse à une demande TouchID, lors de l'entrée dans le sélecteur d'applications ou le centre de contrôle, ou lorsque le UIViewController hébergeant l'application Flutter est en transition. Sur Android, les applications passent à cet état lorsqu'une autre activité est ciblée, telle qu'une application à écran partagé, un appel téléphonique, une application Picture-in-picture (PIP), une boîte de dialogue système ou une autre fenêtre.
  Les applications dans cet état doivent supposer qu'elles peuvent être mises en pause à tout moment.
- **paused** : l'application n'est actuellement pas visible, ne répond pas aux entrées de l'utilisateur et s'exécute en arrière-plan.
- **resumed** : l'application est visible et répond à la saisie de l'utilisateur. Dans cet état, l'application est au premier plan.

Installation des Plugins Pour Android Studio

Dans cette étape, vous allez installer les plugins Flutter et Dart dans Android Studio. Commencez par ouvrrire Android Studio.

Ensuite, cliquez sur Configure, puis sur Plugins.

Dans Marketplace, utilisez la zone de recherche en tapant **Flutter** ensuite **Dart** afin de les installer.

Vous devez redémarrer Android Studio pour voir les plugins nouvellement ajoutés.

Vous avez réussi à configurer Android Studio en tant qu'éditeur de code pour Flutter et Dart.

Accédez à :

Tools -> SDK Manager -> SDK Tools, cochez Android SDK Command-line-Tools et en bas cliquez sur Apply. Attendez que le téléchargement soit terminé et cliquez sur Finish puis Ok.

Configuration de l'émulateur Android

Vous allez utiliser le gestionnaire de périphérique virtuel Android (AVD). AVD Manager est un outil fourni avec Android Studio. Il vous aide à créer et à maintenir des appareils virtuels Android.

Dans le menu, sélectionnez Tools -> AVD Manager. Puis, sélectionnez l'option Create virtual device.

Vous allez avoir la possibilité de configurer un appareil. Appuyer sur le bouton Next .

Vous allez sélectionner la dernière image système d'Android, en cliquant sur Download. Cliquez sur finish.

Cliquez sur la version d'android puis sur Next. Laissez la configuration par défaut et cliquez sur finish.

Vous pouvez vérifier la configuration en lançant le AVD Manager. Pour lancer l'appareil virtuel, appuyez sur le bouton de lecture.

Le périphérique virtuel Android va démarrer sur votre machine.

Pour installer Flutter :

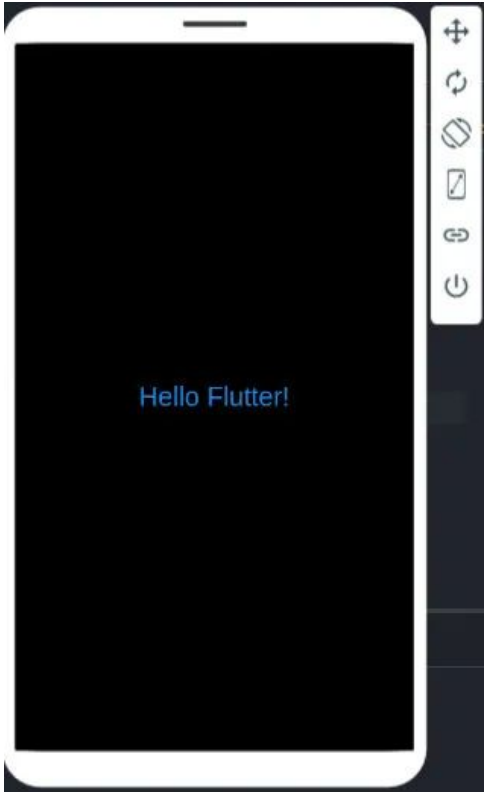
https://docs.flutter.dev/get-started/install/windows

Première application Flutter

Afin de tester votre code vous pouvez utiliser Android Studio :

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

void main() => runApp(
      const Center(
          child: Text("Hello Flutter!",
              textDirection:
TextDirection.ltr,
              style: TextStyle(
                color: Colors.blue,
                fontSize: 25,
              ))),
    );
```


Hello Flutter!

Comme vous le savez, chaque programme Dart doit avoir une fonction `void main()`, et Flutter ne fait pas exception. La méthode `runApp()` est une instance de widget, ce qui en fait la racine de l'arborescence qui démarre notre application.

La méthode `runApp()` prend un widget comme argument. Dans notre cas, `Center` , qui possède un autre widget enfant, `Text`.

Le but principal de `runApp()` est d'attacher le widget donné à l'écran.

Nous avons créé une application Flutter simple qui affiche un texte `"Hello Flutter!"` au centre de l' écran.

Cet exemple nous montre quelques faits intéressants :

Nous avons plusieurs widgets imbriqués, comme Center et Text.

En réalité, les widgets peuvent avoir un ou plusieurs enfants, et cela se produit tout le temps et peut être très profond.

Les widgets Flutter peuvent avoir leurs propres propriétés, comme textDirection, style et color .

Astuce :

Pour l'auto complétion utilisez Ctrl+Espace.

Le widget Text permet d'afficher un texte du côté utilisateur. Il est hautement personnalisable, vous pouvez modifier la couleur et la taille de la police.

textDirection: TextDirection.ltr : définir la direction du texte, ltr qui veut dire left to right.

La structure de répertoires d'une application

Jetons un coup d'œil rapide sur la structure de répertoires d'une application Flutter.

android – dans ce dossier, nous avons tous les fichiers de projet pour l'application Android. Vous pouvez apporter des modifications, ajouter les autorisations requises et le code Android natif.

Le dossier ios – il contient tous les fichiers de projet pour l'application iOS. Vous pouvez ajouter les autorisations requises et ajouter du code iOS natif.

Le dossier lib – c'est le dossier où toute la magie opère. Vous avez un fichier main.dart. Tout votre code Dart est dans ce répertoire. En fait, il sera compilé en code natif (android et iOS).

test – dans ce dossier, vous pouvez écrire des tests pour votre application Flutter.

.gitignore – Je suppose que vous connaissez ce fichier si vous utilisez GIT comme système de contrôle de version pour vos projets. Il contient tous les noms de fichiers et de dossiers qui n'ont pas besoin d'être ajoutés à GIT .

pubspec.lock et pubspec.yaml – ces fichiers contiennent tous les noms de packages requis, leurs versions, les dépendances, le nom de votre application, la version de l'application, les dépendances de votre application, etc.

README – Il s'agit d'un fichier qui contient toutes les informations de base et la description de l'application.

Utiliser plus de widget Flutter

Nous allons voir comment utiliser StatelessWidget ainsi que d’autres widgets et fonctions.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Tutoriel Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Les Widgets Flutter'),
        ),
        body: Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Lorsque vous exécutez un code Flutter une deuxième fois, utilisez le rechargement à chaud, c’est plus rapide.

Autres concepts sur les widgets Flutter

Vous devez d'abord importer le package Flutter,Material Design est un langage visuel standard que Flutter utilise pour fournir des widgets.

La méthode main() utilise la notation de fléche (=>) qui vous permet de raccourcir les fonctions.

Notre application MyApp hérite de StatelessWidget, ce qui en fait également un widget.

Le widget Scaffold (qui veut dire échafaudage), issu de la bibliothèque Material, fournit une **barre d'application** par défaut, un **titre** et un **corps qui contient l'arborescence de widgets de l'écran d'accueil**. La sous-arborescence du widget peut être assez complexe. De plus,Scaffold est généralement utilisé comme sous-widget de MaterialApp, il remplit l'espace disponible et occupe la totalité de la fenêtre ou de l'écran de l'appareil.

En fait, la tâche principale d'un widget consiste à fournir une méthode **build()** qui décrit comment afficher le widget en fonction d'autres widgets de niveau inférieur. Donc, la fonction build()renvoie ce qui doit être affiché à l'écran.

MaterialApp est le widget le plus couramment utilisé pour le développement Flutter, qui est conforme aux concepts de Material Design.

BuildContext est utilisé pour localiser un widget particulier dans une arborescence et chaque widget possède son propre BuidContext. En fait, chaque widget dans Flutter est créé par la méthode build(), qui prend BuildContext comme argument.

Le corps de cet exemple consiste en un widget Center contenant un widget enfant Text. Le widget Center aligne sa sous-arborescence au centre de l'écran.

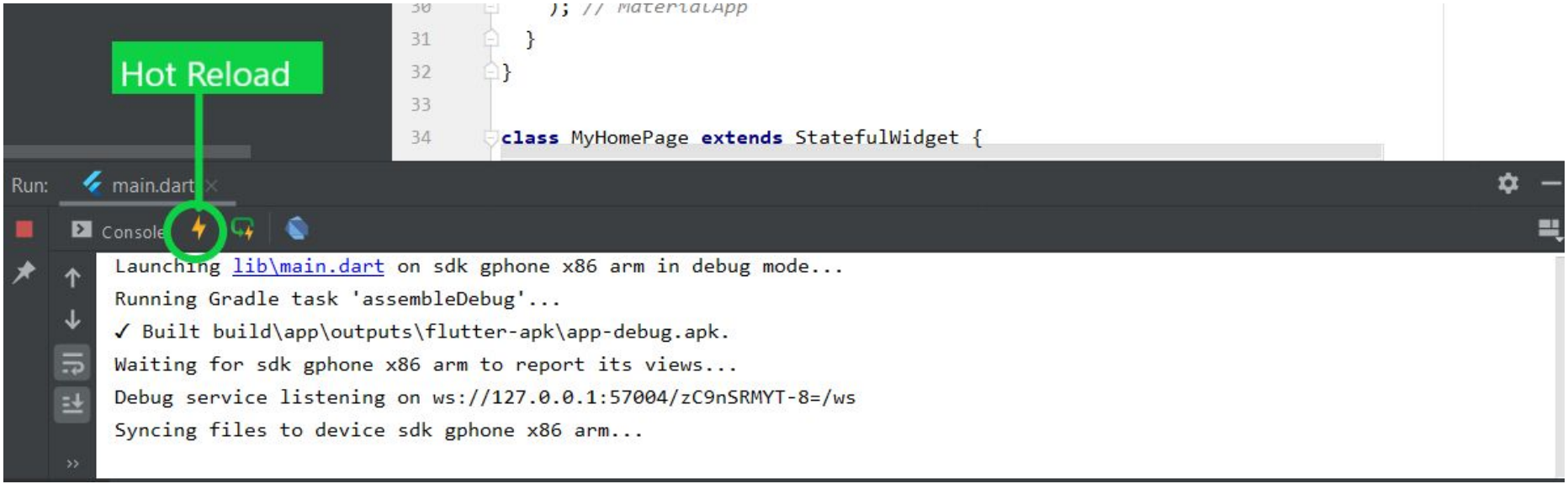
- **Avantages du Hot Reload et Hot Restart**

Lorsque nous construisons une application de fléchettes, son exécution prend du temps pour la première fois. Donc, pour résoudre ce problème, nous avons deux fonctionnalités dans un flutter, à savoir le rechargement à chaud et le redémarrage à chaud, qui aident à réduire le temps d'exécution de notre application une fois que nous l'avons exécutée. Ces deux fonctionnalités permettent de diminuer le temps d'exécution. Ils sont bien meilleurs et plus rapides que le redémarrage par défaut. Il est important de noter qu'il ne peut être utilisé que si vous avez exécuté votre programme une fois.

Rechargement à chaud : Un rechargement à chaud est une excellente fonctionnalité présente dans un flutter. C'est la fonction la plus simple et la plus rapide qui vous aide à appliquer des modifications, à corriger des bogues, à créer des interfaces utilisateur et à ajouter des fonctionnalités. Il faut environ une seconde pour exécuter sa fonctionnalité. En rechargement à chaud, il ne détruit pas l'état préservé. Mais vous ne pouvez pas utiliser un rechargement à chaud une fois que l'application est tuée.

Effectuez un rechargement à chaud :

- Exécutez votre éditeur de flottement à partir de l'application ou à l'aide de l'invite de commande. Nous pouvons utiliser le rechargement à chaud en mode débogage flottant.
- Une fois votre projet flutter créé, apportez quelques modifications à votre code et effectuez un rechargement à chaud.
- Dans Windows, vous pouvez effectuer un rechargement à chaud en utilisant 'ctrl+\' ou en utilisant le bouton de rechargement à chaud. Sur les appareils Mac, vous effectuez un rechargement à chaud à l'aide de 'cmd+s'. Si vous travaillez dans l'invite de commande en utilisant Flutter Run, entrez 'r' pour exécuter.

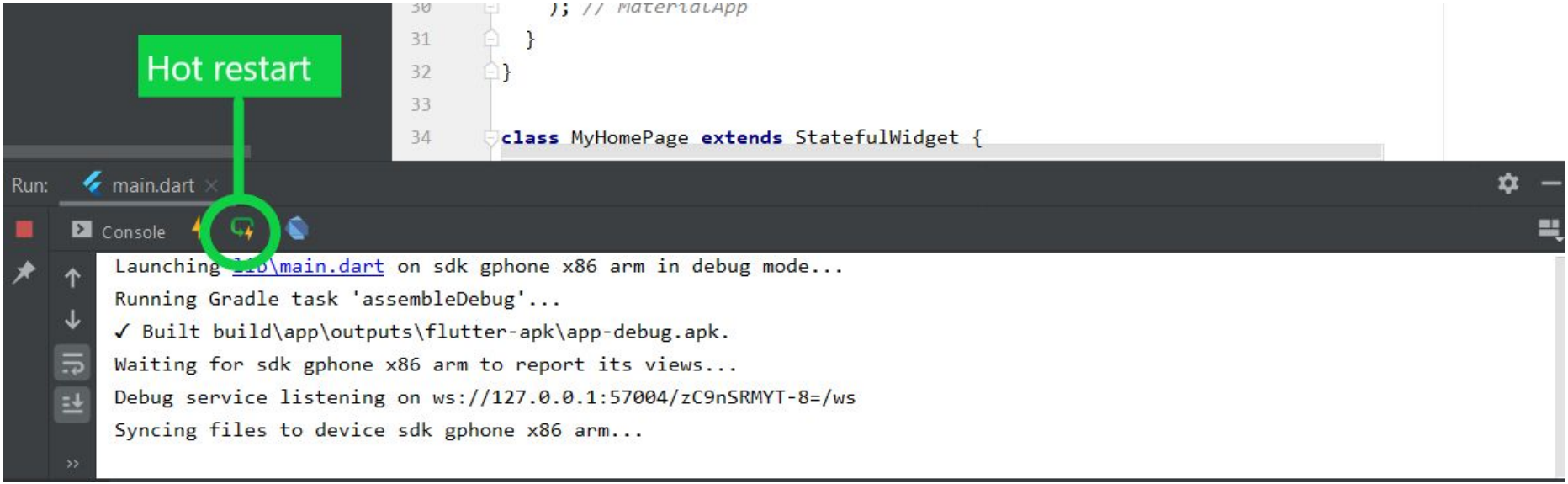

Hot Reload
); // MaterialApp
}
}
class MyHomePage extends StatefulWidget {
Run:
main.dart
Console
Launching lib\main.dart on sdk gphone x86 arm in debug mode...
Running Gradle task 'assembleDebug'...
✓ Built build\app\outputs\flutter-apk\app-debug.apk.
Waiting for sdk gphone x86 arm to report its views...
Debug service listening on ws://127.0.0.1:57004/zC9nSRMYT-8=/ws
Syncing files to device sdk gphone x86 arm...

Redémarrage à chaud :

Un redémarrage à chaud a une fonctionnalité légèrement différente par rapport à un rechargement à chaud. Il est plus rapide que la fonction de redémarrage complet. Il détruit les états préservés de notre application, et le code est à nouveau entièrement compilé et démarre à partir de l'état par défaut. Cela prend plus de temps que le rechargement à chaud mais prend moins de temps que la fonction de redémarrage complet.

Effectuez un redémarrage à chaud :

- Exécutez votre éditeur de flottement à partir de l'application ou à l'aide de l'invite de commande.
- Une fois votre projet Flutter créé, apportez quelques modifications à votre code et effectuez un redémarrage à chaud.
- Vous pouvez effectuer un redémarrage à chaud à l'aide du bouton de rechargement à chaud ou en appuyant sur ctrl+shift+\.


Hot restart
); // MaterialApp
}
}
class MyHomePage extends StatefulWidget {
Run:
main.dart
Console
Launching lib\main.dart on sdk gphone x86 arm in debug mode...
Running Gradle task 'assembleDebug'...
✓ Built build\app\outputs\flutter-apk\app-debug.apk.
Waiting for sdk gphone x86 arm to report its views...
Debug service listening on ws://127.0.0.1:57004/zC9nSRMYT-8=/ws
Syncing files to device sdk gphone x86 arm...

Les principales différences sont les suivantes :

Rechargement à chaud

- Il fonctionne très rapidement par rapport au redémarrage à chaud ou au redémarrage par défaut du flottement.
- Si nous utilisons l'état dans notre application, le rechargement à chaud ne changera pas l'état de l'application.
- Nous effectuons un rechargement à chaud en utilisant la touche ctlr+\.

Redémarrage à chaud

- Il est plus lent que le rechargement à chaud mais plus rapide que le redémarrage par défaut.
- Il ne préserve pas l'état de notre application, il part de l'état initial de notre application.
- Nous effectuons un redémarrage à chaud en utilisant ctrl+shift+\

Rechargement à chaud	Redémarrage à chaud
Le rechargement à chaud nous permet de voir le changement reflété après les corrections de bogues, la création d'interfaces utilisateur et même l'ajout de certaines fonctionnalités à l'application sans exécuter votre application encore et encore.	Le redémarrage à chaud détruit la valeur d'état préservée et les définit sur leur valeur par défaut.
Lorsque le rechargement à chaud est appelé, la machine hôte vérifie le code modifié depuis la dernière compilation et le recompile à nouveau.	Le redémarrage à chaud prend beaucoup plus de temps que le rechargement à chaud.

Le rechargement à chaud ne fonctionne pas lorsque les types énumérés sont remplacés par des classes normales et lorsque les classes sont remplacées par des types énumérés.

Le rechargement à chaud ne fonctionne pas lorsque des types génériques sont modifiés

Le rechargement à chaud est également connu sous le nom de "rechargement à chaud avec état"

Le rechargement à chaud est utile car il permet de gagner du temps en implémentant simplement la fonctionnalité basée sur la classe de construction la plus proche en moins de 10 secondes.

- **Bibliothèque dart:convert et parser du JSON**

dart:convert - décodage et encodage JSON, UTF-8, etc.

La bibliothèque dart:convert (référence API) possède des convertisseurs pour JSON et UTF-8, ainsi que la prise en charge de la création de convertisseurs supplémentaires. JSON est un format de texte simple pour représenter des objets structurés et des collections. UTF-8 est un codage commun à largeur variable qui peut représenter chaque caractère du jeu de caractères Unicode.

Pour utiliser cette bibliothèque, importez dart:convert.

```
import 'dart:convert';
```

Décodage et encodage JSON

Décode une chaîne encodée en JSON en un objet Dart avec `jsonDecode()`:

```
// NOTE: Be sure to use double quotes ("),
// not single quotes ('), inside the JSON string.
// This string is JSON, not Dart.
var jsonString = '''
  [
    {"score": 40},
    {"score": 80}
  ]
''';

var scores = jsonDecode(jsonString);
assert(scores is List);

var firstScore = scores[0];
assert(firstScore is Map);
    assert(firstScore['score'] == 40);
```

Encodez un objet Dart pris en charge dans une chaîne au format JSON avec `jsonEncode()` :

```
var scores = [
  {'score': 40},
  {'score': 80},
  {'score': 100, 'overtime': true, 'special_guest': null}
];

var jsonText = jsonEncode(scores);
assert(jsonText ==
    '[{"score":40},{"score":80},'
        '{"score":100,"overtime":true,'
            '"special_guest":null}]');
```

Seuls les objets de type int, double, String, bool, null, List ou Map (avec des clés de chaîne) sont directement encodables en JSON. Les objets List et Map sont encodés de manière récursive.

Vous avez deux options pour encoder des objets qui ne sont pas directement encodables. La première consiste à invoquer `jsonEncode()` avec un second argument : une fonction qui renvoie un objet directement encodable. Votre deuxième option consiste à omettre le deuxième argument, auquel cas l'encodeur appelle la `toJson()` méthode de l'objet.

Pour plus d'exemples et de liens vers des packages liés à JSON, consultez Utilisation de JSON .

Décodage et encodage des caractères UTF-8

À utiliser `utf8.decode()` pour décoder les octets encodés en UTF8 en une chaîne Dart :

```
List<int> utf8Bytes = [
  0xc3, 0x8e, 0xc3, 0xb1, 0xc5, 0xa3, 0xc3, 0xa9,
  0x72, 0xc3, 0xb1, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3,
  0xae, 0xc3, 0xb6, 0xc3, 0xb1, 0xc3, 0xa5, 0xc4,
  0xbc, 0xc3, 0xae, 0xc5, 0xbe, 0xc3, 0xa5, 0xc5,
  0xa3, 0xc3, 0xae, 0xe1, 0xbb, 0x9d, 0xc3, 0xb1
];

var funnyWord = utf8.decode(utf8Bytes);

    assert(funnyWord == 'Îñţérñåţîöñåļîžåţîờñ');
```

Pour convertir un flux de caractères UTF-8 en une chaîne Dart, spécifiez `utf8.decoder` à la méthode Stream `transform()` :

```
var lines = utf8.decoder.bind(inputStream).transform(const LineSplitter());
try {
  await for (final line in lines) {
    print('Got ${line.length} characters from stream');
  }
  print('file is now closed');
} catch (e) {
  print(e);
   }
```

À utiliser `utf8.encode()` pour encoder une chaîne Dart sous la forme d'une liste d'octets encodés en UTF8 :

```
List<int> encoded = utf8.encode('Îñţérñåţîöñåļîžåţîờñ');

assert(encoded.length == utf8Bytes.length);
for (int i = 0; i < encoded.length; i++) {
  assert(encoded[i] == utf8Bytes[i]);
   }
```

- **API Request**

La récupération de données sur Internet est nécessaire pour la plupart des applications. Heureusement, Dart et Flutter fournissent des outils, tels que le `http`package, pour ce type de travail.

Cette recette utilise les étapes suivantes :

1. Ajoutez le `http`paquet.
2. Effectuez une requête réseau à l'aide du `http`package.
3. Convertissez la réponse en un objet Dart personnalisé.
4. Récupérez et affichez les données avec Flutter.

Ajoutez le `http` paquet

Le `http` package fournit le moyen le plus simple de récupérer des données sur Internet.

Pour installer le `http` package, ajoutez-le à la section des dépendances du `pubspec.yaml`fichier. Vous pouvez trouver la dernière version du `http`package dans pub.dev.

```
dependencies:
  http: <latest_version>
```

Importez le package http.

```
import 'package:http/http.dart' as http;
```

De plus, dans votre fichier AndroidManifest.xml, ajoutez l'autorisation Internet.

```
<!-- Required to fetch data from the internet. -->
<uses-permission android:name="android.permission.INTERNET" />
```

Faire une demande de réseau

Cette recette explique comment récupérer un exemple d'album à partir du JSONPlaceholder à l'aide de la `http.get()`méthode .

```
Future<http.Response> fetchAlbum() {
  return http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));
}
```

La `http.get()`méthode renvoie un `Future`qui contient un `Response`.

- `Future` est une classe Dart de base pour travailler avec des opérations asynchrones. Un objet Future représente une valeur ou une erreur potentielle qui sera disponible à un moment donné dans le futur.
- La `http.Response`classe contient les données reçues d'un appel HTTP réussi.

Convertissez la réponse en un objet Dart personnalisé

Bien qu'il soit facile de faire une requête réseau, travailler avec un raw `Future<http.Response>`n'est pas très pratique. Pour vous faciliter la vie, convertissez le `http.Response`en un objet Dart.

Créer une `Album` classe

Tout d'abord, créez une `Album`classe contenant les données de la requête réseau. Il inclut un constructeur d'usine qui crée un `Album`fichier JSON.

La conversion manuelle de JSON n'est qu'une option. Pour plus d'informations, consultez l'article complet sur JSON et la sérialisation .

```
class Album {
  final int userId;
  final int id;
  final String title;

  const Album({
    required this.userId,
    required this.id,
    required this.title,
  });

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
```

Convertir le `http.Response` en un `Album`

Maintenant, utilisez les étapes suivantes pour mettre à jour la `fetchAlbum()` fonction afin de renvoyer un `Future<Album>`:

1. Convertissez le corps de la réponse en JSON `Map`avec le `dart:convert`package.
2. Si le serveur renvoie une réponse OK avec un code d'état de 200, convertissez le JSON `Map`en `Album` utilisant la `fromJson()`méthode d'usine.
3. Si le serveur ne renvoie pas de réponse OK avec un code d'état de 200, lancez une exception. (Même dans le cas d'une réponse de serveur « 404 Not Found », lancez une exception. Ne retournez pas `null`. Ceci est important lors de l'examen des données dans `snapshot`, comme indiqué ci-dessous.)

```dart
Future<Album> fetchAlbum() async {
  final response = await http
      .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}
```

Récupérez les données

Appelez la `fetchAlbum()`méthode dans les méthodes `initState()`ou `didChangeDependencies()` .

La `initState()`méthode est appelée exactement une fois, puis plus jamais. Si vous souhaitez avoir la possibilité de recharger l'API en réponse à un `InheritedWidget`changement, placez l'appel dans la `didChangeDependencies()`méthode. Voir `State`pour plus de détails.

```
class _MyAppState extends State<MyApp> {
  late Future<Album> futureAlbum;

  @override
  void initState() {
    super.initState();
    futureAlbum = fetchAlbum();
  }
  // ···
}
```

Ce futur est utilisé à l'étape suivante.

Afficher les données

Pour afficher les données à l'écran, utilisez le `FutureBuilder`widget. Le `FutureBuilder`widget est fourni avec Flutter et facilite le travail avec des sources de données asynchrones.

Vous devez fournir deux paramètres :

1. Celui `Future`avec qui vous voulez travailler. Dans ce cas, le futur renvoyé par la `fetchAlbum()`fonction.
2. Une `builder`fonction qui indique à Flutter ce qu'il faut rendre, selon l'état du `Future`: chargement, succès ou erreur.

Notez que `snapshot.hasData`ne revient que `true` lorsque l'instantané contient une valeur de données non nulle.

Comme `fetchAlbum` elle ne peut renvoyer que des valeurs non nulles, la fonction doit lever une exception même dans le cas d'une réponse de serveur « 404 Not Found ». Lancer une exception définit le `snapshot.hasError`to `true` qui peut être utilisé pour afficher un message d'erreur.

Sinon, le spinner sera affiché.

```
FutureBuilder<Album>(
  future: futureAlbum,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text(snapshot.data!.title);
    } else if (snapshot.hasError) {
      return Text('${snapshot.error}');
    }

    // By default, show a loading spinner.
    return const CircularProgressIndicator();
  },
)
```

Pourquoi fetchAlbum() est-il appelé dans initState() ?

Bien que ce soit pratique, il n'est pas recommandé de placer un appel d'API dans une `build()`méthode.

Flutter appelle la `build()`méthode chaque fois qu'elle doit modifier quoi que ce soit dans la vue, et cela se produit étonnamment souvent. La `fetchAlbum()`méthode, si elle est placée à l'intérieur `build()`de , est appelée à plusieurs reprises à chaque reconstruction, ce qui ralentit l'application.

Le stockage du `fetchAlbum()`résultat dans une variable d'état garantit que le `Future`n'est exécuté qu'une seule fois, puis mis en cache pour les reconstructions ultérieures.

Code source :

https://gist.github.com/NizarETH/5f59d6b9cd27cbf9c276fd4dc22abd37

Pour récupérer des données à partir de la plupart des services Web, vous devez fournir une autorisation. Il existe de nombreuses façons de procéder, mais la plus courante utilise peut-être l' `Authorization`en-tête HTTP.

Ajouter des en-têtes d'autorisation

Le `http` package fournit un moyen pratique d'ajouter des en-têtes à vos requêtes. Vous pouvez également utiliser la `HttpHeaders` classe de la `dart:io` bibliothèque.

```
final response = await http.get(
  Uri.parse('https://jsonplaceholder.typicode.com/albums/1'),
  // Send authorization headers to the backend.
  headers: {
    HttpHeaders.authorizationHeader: 'Basic your_api_token_here',
  },
);
```

Analyser JSON en arrière-plan

Par défaut, les applications Dart effectuent tout leur travail sur un seul thread. Dans de nombreux cas, ce modèle simplifie le codage et est suffisamment rapide pour ne pas entraîner de mauvaises performances de l'application ou des animations saccadées, souvent appelées "jank".

Cependant, vous devrez peut-être effectuer un calcul coûteux, tel que l'analyse d'un très grand document JSON. Si ce travail prend plus de 16 millisecondes, vos utilisateurs subissent un jank.

Pour éviter le jank, vous devez effectuer des calculs coûteux comme celui-ci en arrière-plan. Sur Android, cela signifie planifier le travail sur un thread différent. Dans Flutter, vous pouvez utiliser un Isolate . Cette recette utilise les étapes suivantes :

1. Ajoutez le `http`paquet.
2. Effectuez une requête réseau à l'aide du `http`package.
3. Convertissez la réponse en une liste de photos.
4. Déplacez ce travail vers un isolat séparé.

1. Ajoutez le `http` paquet

Tout d'abord, ajoutez le `http` package à votre projet. Le `http`package facilite l'exécution de requêtes réseau, telles que la récupération de données à partir d'un point de terminaison JSON.

```
dependencies:
  http: <latest_version>
```

2. Faire une demande de réseau

Cet exemple explique comment récupérer un document JSON volumineux contenant une liste de 5 000 objets photo à partir de l' API REST JSONPlaceholder , à l'aide de la `http.get()`méthode .

```
Future<http.Response> fetchPhotos(http.Client client) async {
  return client.get(Uri.parse('https://jsonplaceholder.typicode.com/photos'));
}
```

3. Analysez et convertissez le JSON en une liste de photos

Ensuite, en suivant les instructions de la recette Récupérer les données de la recette Internet , convertissez le `http.Response`en une liste d'objets Dart. Cela rend les données plus faciles à utiliser.

Créer une `Photo` classe

Tout d'abord, créez une `Photo`classe contenant des données sur une photo. Incluez une `fromJson()`méthode de fabrique pour faciliter la création d'un `Photo`démarrage avec un objet JSON.

```dart
class Photo {
  final int albumId;
  final int id;
  final String title;
  final String url;
  final String thumbnailUrl;

  const Photo({
    required this.albumId,
    required this.id,
    required this.title,
    required this.url,
    required this.thumbnailUrl,
```

Convertir la réponse en une liste de photos

Maintenant, utilisez les instructions suivantes pour mettre à jour la `fetchPhotos()`fonction afin qu'elle renvoie un `Future<List<Photo>>`:

1. Créez une `parsePhotos()`fonction qui convertit le corps de la réponse en un fichier `List<Photo>`.
2. Utilisez la `parsePhotos()`fonction dans la `fetchPhotos()`fonction.

```
// A function that converts a response body into a List<Photo>.
List<Photo> parsePhotos(String responseBody) {
  final parsed = jsonDecode(responseBody).cast<Map<String, dynamic>>();

  return parsed.map<Photo>((json) => Photo.fromJson(json)).toList();
}

Future<List<Photo>> fetchPhotos(http.Client client) async {
  final response = await client
      .get(Uri.parse('https://jsonplaceholder.typicode.com/photos'));

  // Use the compute function to run parsePhotos in a separate isolate.
  return parsePhotos(response.body);
}
```

Déplacez ce travail vers un isolat séparé

Si vous exécutez la `fetchPhotos()` fonction sur un appareil plus lent, vous remarquerez peut-être que l'application se fige pendant un bref instant lorsqu'elle analyse et convertit le JSON. C'est de la merde, et vous voulez vous en débarrasser.

Vous pouvez supprimer le jank en déplaçant l'analyse et la conversion vers un isolat d'arrière-plan à l'aide de la `compute()` fonction fournie par Flutter. La `compute()` fonction exécute des fonctions coûteuses dans un isolat d'arrière-plan et renvoie le résultat. Dans ce cas, exécutez la `parsePhotos()` fonction en arrière-plan.

```
Future<List<Photo>> fetchPhotos(http.Client client) async {
  final response = await client
      .get(Uri.parse('https://jsonplaceholder.typicode.com/photos'));

  // Use the compute function to run parsePhotos in a separate isolate.
  return compute(parsePhotos, response.body);
}
```

Notes sur le travail avec des isolats

Les isolats communiquent en passant des messages dans les deux sens. Ces messages peuvent être des valeurs primitives, telles que `null`, `num`, `bool`, `double`, ou `String`, ou des objets simples tels que `List<Photo>`dans cet exemple.

Vous pouvez rencontrer des erreurs si vous essayez de transmettre des objets plus complexes, tels qu'un `Future`ou `http.Response`entre des isolats.

Comme solution alternative, consultez les packages `worker_manager`ou `workmanager`pour le traitement en arrière-plan.

Envoyer des données sur Internet

L'envoi de données sur Internet est nécessaire pour la plupart des applications. Le `http`paquet a également cela couvert.

Cette recette utilise les étapes suivantes :

1. Ajoutez le `http`paquet.
2. Envoyez des données à un serveur à l'aide du `http`package.
3. Convertissez la réponse en un objet Dart personnalisé.
4. Obtenez une `title`entrée de l'utilisateur.
5. Affichez la réponse à l'écran.

Ajoutez le `http` paquet

Pour installer le `http` package, ajoutez-le à la section des dépendances du `pubspec.yaml`fichier. Vous pouvez trouver la dernière version du `http`package sur pub.dev.

```
dependencies:
  http: <latest_version>
```

Importez le `http` paquet.

```
import 'package:http/http.dart' as http;
```

Si vous développez pour Android, ajoutez l'autorisation suivante dans la balise manifest du fichier AndroidManifest.xml situé dans android/app/src/main.

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

Envoi de données au serveur

Cette recette explique comment créer un `Album` en envoyant un titre d'album au JSONPlaceholder à l'aide de la `http.post()`méthode .

```
Future<http.Response> createAlbum(String title) {
  return http.post(
    Uri.parse('https://jsonplaceholder.typicode.com/albums'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'title': title,
    }),
  );
}
```

La `http.post()` méthode renvoie un `Future` qui contient un `Response`.

- `Future` est une classe Dart de base pour travailler avec des opérations asynchrones. Un objet Future représente une valeur ou une erreur potentielle qui sera disponible à un moment donné dans le futur.
- La `http.Response` classe contient les données reçues d'un appel HTTP réussi.
- La `createAlbum()` méthode prend un argument `title` qui est envoyé au serveur pour créer un fichier `Album`.

3. Convertissez le `http.Response` en un objet Dart personnalisé

Bien qu'il soit facile de faire une requête réseau, travailler avec un raw `Future<http.Response>` n'est pas très pratique. Pour vous faciliter la vie, convertissez le `http.Response`en un objet Dart.

Créer une classe Album

Tout d'abord, créez une `Album` classe contenant les données de la requête réseau. Il inclut un constructeur d'usine qui crée un `Album` fichier JSON.

La conversion manuelle de JSON n'est qu'une option. Pour plus d'informations, consultez l'article complet sur JSON et la sérialisation .

```
class Album {
  final int id;
  final String title;

  const Album({required this.id, required this.title});

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      id: json['id'],
      title: json['title'],
    );
  }
```

Convertir le `http.Response` en un `Album`

Utilisez les étapes suivantes pour mettre à jour la `createAlbum()` fonction afin de renvoyer un `Future<Album>`:

1. Convertissez le corps de la réponse en JSON `Map` avec le `dart:convert`package.
2. Si le serveur renvoie une `CREATED`réponse avec un code d'état de 201, convertissez le JSON `Map`en `Album` utilisant la `fromJson()`méthode d'usine.
3. Si le serveur ne renvoie pas de `CREATED`réponse avec un code d'état de 201, lancez une exception. (Même dans le cas d'une réponse de serveur « 404 Not Found », lancez une exception. Ne retournez pas `null`. Ceci est important lors de l'examen des données dans `snapshot`, comme indiqué ci-dessous.)

```
Future<Album> createAlbum(String title) async {
  final response = await http.post(
    Uri.parse('https://jsonplaceholder.typicode.com/albums'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'title': title,
    }),
  );

  if (response.statusCode == 201) {
    // If the server did return a 201 CREATED response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 201 CREATED response,
    // then throw an exception.
    throw Exception('Failed to create album.');
  }
}
```

Vous avez maintenant une fonction qui envoie le titre à un serveur pour créer un album.

Obtenez un titre à partir de l'entrée de l'utilisateur

Ensuite, créez un `TextField` pour entrer un titre et un `ElevatedButton`pour envoyer des données au serveur. Définissez également a `TextEditingController` pour lire l'entrée utilisateur à partir d'un fichier `TextField`.

Lorsque la `ElevatedButton`touche est enfoncée, la `_futureAlbum` est définie sur la valeur renvoyée par la `createAlbum()`méthode.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    TextField(
      controller: _controller,
      decoration: const InputDecoration(hintText: 'Enter Title'),
    ),
    ElevatedButton(
      onPressed: () {
        setState(() {
          _futureAlbum = createAlbum(_controller.text);
        });
      },
      child: const Text('Create Data'),
    ),
  ],
)
```

En appuyant sur le bouton Créer des données , effectuez la requête réseau, qui envoie les données du `TextField`au serveur sous forme de `POST`requête. Le futur, `_futureAlbum`, est utilisé à l'étape suivante.

Affichez la réponse à l'écran

Pour afficher les données à l'écran, utilisez le `FutureBuilder` widget. Le `FutureBuilder` widget est fourni avec Flutter et facilite le travail avec des sources de données asynchrones. Vous devez fournir deux paramètres :

1. Celui `Future`avec qui vous voulez travailler. Dans ce cas, le futur renvoyé par la `createAlbum()`fonction.
2. Une `builder`fonction qui indique à Flutter ce qu'il faut rendre, selon l'état du `Future`: chargement, succès ou erreur.

Notez que `snapshot.hasData`ne revient que `true`lorsque l'instantané contient une valeur de données non nulle. C'est pourquoi la `createAlbum()`fonction doit lever une exception même dans le cas d'une réponse de serveur "404 Not Found". Si `createAlbum()`renvoie `null`, alors `CircularProgressIndicators`'affiche indéfiniment.

```
FutureBuilder<Album>(
  future: _futureAlbum,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text(snapshot.data!.title);
    } else if (snapshot.hasError) {
      return Text('${snapshot.error}');
    }

    return const CircularProgressIndicator();
  },
)
```

Code source

https://gist.github.com/NizarETH/2cadcbba5f44df61698bd94c3e4b79de

Mettre à jour les données via Internet

La mise à jour des données via Internet est nécessaire pour la plupart des applications. Le `http`paquet a tout ce qu'il faut !

Cette recette utilise les étapes suivantes :

1. Ajoutez le `http`paquet.
2. Mettez à jour les données sur Internet à l'aide du `http`package.
3. Convertissez la réponse en un objet Dart personnalisé.
4. Obtenez les données sur Internet.
5. Mettez à jour l'existant `title`à partir de l'entrée de l'utilisateur.
6. Mettez à jour et affichez la réponse à l'écran.

1. Ajoutez le `http`paquet

Pour installer le `http` package, ajoutez-le à la section des dépendances du `pubspec.yaml` fichier. Vous pouvez trouver la dernière version du `http`package sur pub.dev.

```
dependencies:
  http: <latest_version>
```

Importez le `http`paquet.

```
import 'package:http/http.dart' as http;
```

2. Mise à jour des données sur Internet à l'aide du `http` package

Cette recette explique comment mettre à jour un titre d'album vers JSONPlaceholder à l'aide de la `http.put()` méthode .

```
Future<http.Response> updateAlbum(String title) {
  return http.put(
    Uri.parse('https://jsonplaceholder.typicode.com/albums/1'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'title': title,
    }),
  );
}
```

La `http.put()`méthode renvoie un `Future`qui contient un `Response`.

- `Future`est une classe Dart de base pour travailler avec des opérations asynchrones. Un `Future`objet représente une valeur ou une erreur potentielle qui sera disponible à un moment donné dans le futur.
- La `http.Response`classe contient les données reçues d'un appel HTTP réussi.
- La `updateAlbum()`méthode prend un argument, `title`, qui est envoyé au serveur pour mettre à jour le `Album`.

3. Convertissez le `http.Response` en un objet Dart personnalisé

Bien qu'il soit facile de faire une requête réseau, travailler avec un raw `Future<http.Response>` n'est pas très pratique. Pour vous faciliter la vie, convertissez le `http.Response`en un objet Dart.

Créer une classe Album

Tout d'abord, créez une `Album`classe contenant les données de la requête réseau. Il inclut un constructeur d'usine qui crée un `Album`fichier JSON.

La conversion manuelle de JSON n'est qu'une option. Pour plus d'informations, consultez l'article complet sur JSON et la sérialisation .

```dart
class Album {
  final int id;
  final String title;

  const Album({required this.id, required this.title});

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      id: json['id'],
      title: json['title'],
    );
  }
}
```

Convertir le `http.Response` en un `Album`

Maintenant, utilisez les étapes suivantes pour mettre à jour la `updateAlbum()` fonction afin de renvoyer un `Future<Album>`:

1. Convertissez le corps de la réponse en JSON `Map`avec le `dart:convert`package.
2. Si le serveur renvoie une `UPDATED`réponse avec un code d'état de 200, convertissez le JSON `Map`en `Album` utilisant la `fromJson()`méthode d'usine.
3. Si le serveur ne renvoie pas `UPDATED`de réponse avec un code d'état de 200, lancez une exception. (Même dans le cas d'une réponse de serveur « 404 Not Found », lancez une exception. Ne retournez pas `null`. Ceci est important lors de l'examen des données dans `snapshot`, comme indiqué ci-dessous.)

```dart
Future<Album> updateAlbum(String title) async {
  final response = await http.put(
    Uri.parse('https://jsonplaceholder.typicode.com/albums/1'),
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      'title': title,
    }),
  );

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to update album.');
  }
}
```

Vous avez maintenant une fonction qui met à jour le titre d'un album.

4. Récupérez les données sur Internet

Obtenez les données sur Internet avant de pouvoir les mettre à jour. Pour un exemple complet, consultez la recette Fetch data .

```
Future<Album> fetchAlbum() async {
  final response = await http.get(
    Uri.parse('https://jsonplaceholder.typicode.com/albums/1'),);
  if (response.statusCode == 200) {
    return Album.fromJson(jsonDecode(response.body));
  } else {
    throw Exception('Failed to load album');
  }
}
```

Idéalement, vous utiliserez cette méthode pour définir `_futureAlbum` pendant `initState` la récupération des données sur Internet.

Mettre à jour le titre existant à partir de l'entrée de l'utilisateur

Créez un `TextField` pour entrer un titre et un `ElevatedButton` pour mettre à jour les données sur le serveur. Définissez également a `TextEditingController`pour lire l'entrée utilisateur à partir d'un fichier `TextField`.

Lorsque la `ElevatedButton`touche est enfoncée, la `_futureAlbum` est définie sur la valeur renvoyée par la `updateAlbum()` méthode.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Padding(
      padding: const EdgeInsets.all(8.0),
      child: TextField(
        controller: _controller,
        decoration: const InputDecoration(hintText: 'Enter Title'),
      ),
```

Affichez la réponse à l'écran

Pour afficher les données à l'écran, utilisez le `FutureBuilder` widget. Le `FutureBuilder` widget est fourni avec Flutter et facilite le travail avec des sources de données asynchrones. Vous devez fournir deux paramètres :

1. Celui `Future`avec qui vous voulez travailler. Dans ce cas, le futur renvoyé par la `updateAlbum()`fonction.
2. Une `builder`fonction qui indique à Flutter ce qu'il faut rendre, selon l'état du `Future`: chargement, succès ou erreur.

Notez que `snapshot.hasData`ne revient que `true`lorsque l'instantané contient une valeur de données non nulle. C'est pourquoi la `updateAlbum`fonction doit lever une exception même dans le cas d'une réponse de serveur "404 Not Found". Si `updateAlbum`renvoie `null`alors `CircularProgressIndicator`s'affichera indéfiniment.

```
FutureBuilder<Album>(
  future: _futureAlbum,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text(snapshot.data!.title);
    } else if (snapshot.hasError) {
      return Text('${snapshot.error}');
    }

    return const CircularProgressIndicator();
  },
);
```

- **Utiliser SQLite et le plugin sqflite**

Si vous écrivez 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 `sqflite`plugin disponible sur pub.dev. Cette recette montre les bases de l'utilisation `sqflite` pour insérer, lire, mettre à jour et supprimer des données sur divers chiens.

Si vous débutez avec SQLite et les instructions SQL, consultez le didacticiel SQLite pour apprendre les bases avant de terminer cette recette.

Cette recette utilise les étapes suivantes :

1. Ajoutez les dépendances.
2. Définir le `Dog` modèle de données.
3. Ouvrez la base de données.
4. Créez le `dogs` tableau.
5. Insérez un `Dog` dans la base de données.
6. Récupérez la liste des chiens.
7. Mettre à jour un `Dog` dans la base de données.
8. Supprimer un `Dog` de la base de données.

1. Ajouter les dépendances

Pour travailler avec des bases de données SQLite, importez les packages `sqflite`et .`path`

- Le `sqflite` package fournit des classes et des fonctions pour interagir avec une base de données SQLite.
- Le `path` package fournit des fonctions pour définir l'emplacement de stockage de la base de données sur le disque.

```
dependencies:

  flutter:

    sdk: flutter

  sqflite:

  path:
```

Assurez-vous d'importer les packages dans le fichier dans lequel vous allez travailler.

```
import 'dart:async';


import 'package:flutter/widgets.dart';

import 'package:path/path.dart';

import 'package:sqflite/sqflite.dart';
```

Définir le modèle de données Dog

Avant de créer la table pour stocker les informations sur les Chiens, prenez quelques instants pour définir les données qui doivent être stockées. Pour cet exemple, définissez une classe Dog qui contient trois éléments de données : A unique `id`, the `name` et the `age` de chaque chien.

```
class Dog {
  final int id;
  final String name;
  final int age;
   const Dog({
    required this.id,
    required this.name,
    required this.age,
  });
}
```

Ouvrez la base de données

Avant de lire et d'écrire des données dans la base de données, ouvrez une connexion à la base de données. Cela implique deux étapes :

1. Définissez le chemin d'accès au fichier de base de données à l'aide `getDatabasesPath()` du `sqflite` package, combiné avec la `join` fonction du `path` package.
2. Ouvrez la base de données avec la `openDatabase()` fonction de `sqflite`.

Remarque : Pour utiliser le mot-clé `await`, le code doit être placé à l'intérieur d'une `async`fonction. Vous devez placer toutes les fonctions de table suivantes à l'intérieur de `void main() async {}`.

```
// Avoid errors caused by flutter upgrade.
// Importing 'package:flutter/widgets.dart' is required.
WidgetsFlutterBinding.ensureInitialized();
// Open the database and store the reference.
final database = openDatabase(
  // Set the path to the database. Note: Using the `join` function from the
  // `path` package is best practice to ensure the path is correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
);
```

Créez le `dogs` tableau

Ensuite, créez une table pour stocker des informations sur divers chiens. Pour cet exemple, créez une table appelée `dogs` qui définit les données pouvant être stockées. Chacun `Dog` contient un `id`, `name` et `age`. Par conséquent, ceux-ci sont représentés par trois colonnes dans le `dogs` tableau.

1. Le `id` est un Dart `int` et est stocké en tant que type de `INTEGER`données SQLite. Il est également recommandé d'utiliser un `id` comme clé primaire de la table afin d'améliorer les temps de requête et de mise à jour.
2. Le `name` est un Dart `String`et est stocké en tant que type de `TEXT`données SQLite.
3. Le `age` est également un Dart `int`et est stocké en tant que `INTEGER` Datatype.

Pour plus d'informations sur les types de données disponibles pouvant être stockés dans une base de données SQLite, consultez la documentation officielle des types de données SQLite .

```dart
final database = openDatabase(
  // Set the path to the database. Note: Using the `join` function from the
  // `path` package is best practice to ensure the path is correctly
  // constructed for each platform.
  join(await getDatabasesPath(), 'doggie_database.db'),
  // When the database is first created, create a table to store dogs.
  onCreate: (db, version) {
    // Run the CREATE TABLE statement on the database.
    return db.execute(
      'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
    );
  },
  // Set the version. This executes the onCreate function and provides a
  // path to perform database upgrades and downgrades.
  version: 1,
);
```

Insérez un chien dans la base de données

Maintenant que vous avez une base de données avec une table appropriée pour stocker des informations sur divers chiens, il est temps de lire et d'écrire des données.

Tout d'abord, insérez un `Dog`dans le `dogs`tableau. Cela implique deux étapes :

1. Convertissez le `Dog`en un`Map`
2. Utilisez la `insert()`méthode pour stocker le `Map`dans la `dogs`table.

```dart
class Dog {
  final int id;
  final String name;
  final int age;

  const Dog({
    required this.id,
    required this.name,
    required this.age,
  });

  // Convert a Dog into a Map. The keys must correspond to the names of the
  // columns in the database.
  Map<String, dynamic> toMap() {
    return {
      'id': id,
      'name': name,
      'age': age,
    };
  }

  @override
  String toString() {
    return 'Dog{id: $id, name: $name, age: $age}';
```

```dart
// Define a function that inserts dogs into the database
Future<void> insertDog(Dog dog) async {
  // Get a reference to the database.
  final db = await database;

  // Insert the Dog into the correct table. You might also specify the
  // `conflictAlgorithm` to use in case the same dog is inserted twice.
  //
  // In this case, replace any previous data.
  await db.insert(
    'dogs',
    dog.toMap(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}
```

```dart
// Create a Dog and add it to the dogs table
var fido = const Dog(
  id: 0,
  name: 'Fido',
  age: 35,
);

await insertDog(fido);
```

Récupérer la liste des Chiens

Maintenant qu'un `Dog` est stocké dans la base de données, interrogez la base de données pour un chien spécifique ou une liste de tous les chiens. Cela implique deux étapes :

1. Exécutez un `query`contre la `dogs`table. Cela renvoie un `List<Map>`.
2. Convertissez le `List<Map>`en un `List<Dog>`.

```dart
// A method that retrieves all the dogs from the dogs table.
Future<List<Dog>> dogs() async {
  // Get a reference to the database.
  final db = await database;

  // Query the table for all The Dogs.
  final List<Map<String, dynamic>> maps = await db.query('dogs');

  // Convert the List<Map<String, dynamic> into a List<Dog>.
  return List.generate(maps.length, (i) {
    return Dog(
      id: maps[i]['id'],
      name: maps[i]['name'],
      age: maps[i]['age'],
    );
  });
}
// Now, use the method above to retrieve all the dogs.
print(await dogs()); // Prints a list that include Fido.
```

7. Mettre à jour un `Dog` dans la base de données

Après avoir inséré des informations dans la base de données, vous souhaiterez peut-être mettre à jour ces informations ultérieurement. Vous pouvez le faire en utilisant la `update()` méthode de la `sqflite`bibliothèque.

Cela implique deux étapes :

1. Convertissez le chien en carte.
2. Utilisez une `where` clause pour vous assurer de mettre à jour le bon chien.

```dart
Future<void> updateDog(Dog dog) async {
  // Get a reference to the database.
  final db = await database;

  // Update the given Dog.
  await db.update(
    'dogs',
    dog.toMap(),
    // Ensure that the Dog has a matching id.
    where: 'id = ?',
    // Pass the Dog's id as a whereArg to prevent SQL injection.
    whereArgs: [dog.id],
  );
}
```

```dart
// Update Fido's age and save it to the database.
fido = Dog(
  id: fido.id,
  name: fido.name,
  age: fido.age + 7,
);
await updateDog(fido);

// Print the updated results.
print(await dogs()); // Prints Fido with age 42.
```

Avertissement : Utilisez toujours `whereArgs` pour passer des arguments à une `where` instruction. Cela permet de se prémunir contre les attaques par injection SQL.

N'utilisez pas d'interpolation de chaîne, telle que `where: "id = ${dog.id}"`!

8. Supprimer un `Dog` de la base de données

En plus d'insérer et de mettre à jour des informations sur les chiens, vous pouvez également supprimer des chiens de la base de données. Pour supprimer des données, utilisez la `delete()`méthode de la `sqflite`bibliothèque.

Dans cette section, créez une fonction qui prend un identifiant et supprime le chien avec un identifiant correspondant de la base de données. Pour que cela fonctionne, vous devez fournir une `where` clause pour limiter les enregistrements supprimés.

```dart
Future<void> deleteDog(int id) async {
  // Get a reference to the database.
  final db = await database;

  // Remove the Dog from the database.
  await db.delete(
    'dogs',
    // Use a `where` clause to delete a specific dog.
    where: 'id = ?',
    // Pass the Dog's id as a whereArg to prevent SQL injection.
    whereArgs: [id],
  );
}
```

Exemple

Pour exécuter l'exemple :

1. Créez un nouveau projet Flutter.
2. Ajoutez les packages `sqflite`et à votre fichier .`pathpubspec.yaml`
3. Collez le code suivant dans un nouveau fichier nommé `lib/db_test.dart`.
4. Exécutez le code avec `flutter run lib/db_test.dart`.

Code source :

https://gist.github.com/NizarETH/7bf185b718b7445a859b96ccf27f34d8

- **Stateful et stateless widgets**
- **Méthode setState()**
- **Widget interactif**

Ajouter de l'interactivité à votre application Flutter

Ce que vous apprendrez

- Comment répondre aux taps.
- Comment créer un widget personnalisé.
- La différence entre les widgets sans état et avec état.

Comment modifiez-vous votre application pour la faire réagir aux entrées de l'utilisateur ? Dans ce didacticiel, vous allez ajouter de l'interactivité à une application qui ne contient que des widgets non interactifs. Plus précisément, vous allez modifier une icône pour la rendre accessible en créant un widget avec état personnalisé qui gère deux widgets sans état.

Le didacticiel sur les mises en page de construction vous a montré comment créer la mise en page pour la capture d'écran suivante.

Widgets avec et sans état

Un widget est soit avec état, soit sans état. Si un widget peut changer, lorsqu'un utilisateur interagit avec lui, par exemple, il est avec état.

Un widget *sans état* ne change jamais. `Icon`, `IconButton`et `Text`sont des exemples de widgets sans état. Sous-classe de widgets sans état `StatelessWidget`.

Un *widget* avec état est dynamique : par exemple, il peut changer d'apparence en réponse à des événements déclenchés par des interactions de l'utilisateur ou lorsqu'il reçoit des données. `Checkbox`, `Radio`, `Slider`, `InkWell`, `Form`et `TextField` sont des exemples de widgets avec état. Sous-classe de widgets avec état `StatefulWidget`.

L'état d'un widget est stocké dans un `State`objet, séparant l'état du widget de son apparence. L'état se compose de valeurs qui peuvent changer, comme la valeur actuelle d'un curseur ou si une case est cochée. Lorsque l'état du widget change, l'objet d'état appelle `setState()`, indiquant au framework de redessiner le widget.

Étape 1 : Décidez quel objet gère l'état du widget

L'état d'un widget peut être géré de plusieurs manières, mais dans notre exemple, le widget lui-même, `FavoriteWidget`, gérera son propre état. Dans cet exemple, basculer l'étoile est une action isolée qui n'affecte pas le widget parent ou le reste de l'interface utilisateur, de sorte que le widget peut gérer son état en interne.

En savoir plus sur la séparation du widget et de l'état, et sur la manière dont l'état peut être géré, dans Gestion de l'état .

Étape 2 : Sous-classe StatefulWidget

La `FavoriteWidget`classe gère son propre état, elle se substitue donc `createState()`pour créer un `State` objet. Le framework appelle `createState()` quand il veut construire le widget. Dans cet exemple, `createState()`renvoie une instance de `_FavoriteWidgetState`, que vous implémenterez à l'étape suivante.

lib/main.dart (FavoriteWidget)

```
class FavoriteWidget extends StatefulWidget {
  const FavoriteWidget({super.key});


  @override
  State<FavoriteWidget> createState() => _FavoriteWidgetState();
}
```

Étape 3 : État de la sous-classe

La `_FavoriteWidgetState`classe stocke les données mutables qui peuvent changer au cours de la durée de vie du widget. Lorsque l'application est lancée pour la première fois, l'interface utilisateur affiche une étoile rouge solide, indiquant que le lac a le statut « favori », ainsi que 41 j'aime. Ces valeurs sont stockées dans les champs `_isFavorited`et :`_favoriteCount`

lib/main.dart (champs _FavoriteWidgetState)

```
class _FavoriteWidgetState extends State<FavoriteWidget> {

  bool _isFavorited = true;

  int _favoriteCount = 41;
}
```

La classe définit également une `build()`méthode, qui crée une ligne contenant un rouge `IconButton`, et `Text`. Vous utilisez `IconButton`(au lieu de `Icon`) car il a une `onPressed`propriété qui définit la fonction de rappel (`_toggleFavorite`) pour gérer un tap. Vous définirez ensuite la fonction de rappel.

```
class _FavoriteWidgetState extends State<FavoriteWidget> {
  // ···
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        Container(
          padding: const EdgeInsets.all(0),
          child: IconButton(
```

Étape 4 : Branchez le widget avec état dans l'arborescence des widgets

Ajoutez votre widget avec état personnalisé à l'arborescence des widgets dans la `build()`méthode de l'application. Tout d'abord, localisez le code qui crée le `Icon` et `Text` et supprimez-le. Au même emplacement, créez le widget avec état :

Création d'un widget avec état

```
@@ -10,2 +5,2 @@
  la classe  MyApp  étend  StatelessWidget  {
    const MonApp({ super .key});
@@ -40,11 +35,7 @@
                ],
              ),
            ),
-           Icône(
-             Icons.star,
-              couleur : Couleurs.rouge[ 500 ],
-           ),
-  texte const ( '41' ),
+           const FavoriteWidget(),
          ],
        ),
      );
```

État de gestion

À quoi ça sert?

- Il existe différentes approches pour gérer l'état.
- En tant que concepteur de widget, vous choisissez l'approche à utiliser.
- En cas de doute, commencez par gérer l'état dans le widget parent.

Qui gère l'état du widget avec état ? Le widget lui-même ? Le widget parent ? Tous les deux? Un autre objet ? La réponse est… ça dépend. Il existe plusieurs façons valides de rendre votre widget interactif. En tant que concepteur de widget, vous prenez la décision en fonction de la manière dont vous vous attendez à ce que votre widget soit utilisé. Voici les façons les plus courantes de gérer l'état :

- Le widget gère son propre état
- Le parent gère l'état du widget
- Une approche mixte

Comment décidez-vous de l'approche à utiliser ? Les principes suivants devraient vous aider à décider :

- Si l'état en question est des données utilisateur, par exemple le mode coché ou non coché d'une case à cocher, ou la position d'un curseur, alors l'état est mieux géré par le widget parent.
- Si l'état en question est esthétique, par exemple une animation, alors l'état est mieux géré par le widget lui-même.

En cas de doute, commencez par gérer l'état dans le widget parent.

Nous donnerons des exemples des différentes manières de gérer l'état en créant trois exemples simples : TapboxA, TapboxB et TapboxC. Les exemples fonctionnent tous de la même manière - chacun crée un conteneur qui, lorsqu'il est tapé, bascule entre une case verte ou grise. Le `_active`booléen détermine la couleur : vert pour actif ou gris pour inactif.

widgets interactifs

Flutter propose une variété de boutons et de widgets interactifs similaires. La plupart de ces widgets implémentent les directives de conception matérielle , qui définissent un ensemble de composants avec une interface utilisateur avisée.

Si vous préférez, vous pouvez utiliser `GestureDetector`pour créer de l'interactivité dans n'importe quel widget personnalisé. Vous pouvez trouver des exemples de `GestureDetector`dans l'état Gestion . En savoir plus sur les robinets `GestureDetector` à poignée , une recette du livre de recettes Flutter .

Si vous préférez, vous pouvez utiliser `GestureDetector`pour créer de l'interactivité dans n'importe quel widget personnalisé. Vous pouvez trouver des exemples de `GestureDetector`dans l'état Gestion . En savoir plus sur les robinets `GestureDetector` à poignée , une recette du livre de recettes Flutter .

Astuce : Flutter fournit également un ensemble de widgets de style iOS appelés `Cupertino`.
Lorsque vous avez besoin d'interactivité, il est plus simple d'utiliser l'un des widgets préfabriqués. Voici une liste partielle :

Widgets standards

- `Form`
- `FormField`

- **Navigation entre deux écrans : MaterialPageRoute, méthodes push() et pop()**

Naviguer vers un nouvel écran et revenir en arrière

La plupart des applications contiennent plusieurs écrans pour afficher différents types d'informations. Par exemple, une application peut avoir un écran qui affiche des produits. Lorsque l'utilisateur appuie sur l'image d'un produit, un nouvel écran affiche des détails sur le produit.

Terminologie : Dans Flutter, les *écrans* et les *pages* sont appelés *routes* . Le reste de cette recette fait référence aux itinéraires.
Sous Android, un itinéraire équivaut à une activité. Dans iOS, une route équivaut à un ViewController. Dans Flutter, une route n'est qu'un widget.

Cette recette utilise le `Navigator`pour naviguer vers un nouvel itinéraire.

Les quelques sections suivantes montrent comment naviguer entre deux routes, en suivant ces étapes :

1. Créez deux itinéraires.
2. Accédez au deuxième itinéraire à l'aide de Navigator.push().
3. Revenez au premier itinéraire en utilisant Navigator.pop().

1. Créez deux itinéraires

Tout d'abord, créez deux itinéraires avec lesquels travailler. Comme il s'agit d'un exemple de base, chaque route ne contient qu'un seul bouton. Toucher le bouton sur le premier itinéraire permet de naviguer vers le deuxième itinéraire. Appuyez sur le bouton sur le deuxième itinéraire pour revenir au premier itinéraire.

Tout d'abord, configurez la structure visuelle :

```
class FirstRoute extends StatelessWidget {
  const FirstRoute({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            // Navigate to second route when tapped.
```

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


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate back to first route when tapped.
```

Accédez à la deuxième route à l'aide de Navigator.push()

Pour passer à une nouvelle route, utilisez la `Navigator.push()` méthode. La `push()`méthode ajoute un `Route`à la pile de routes gérées par le `Navigator`. D'où vient le `Route`? Vous pouvez créer le vôtre ou utiliser un `MaterialPageRoute`, ce qui est utile car il effectue la transition vers le nouvel itinéraire à l'aide d'une animation spécifique à la plate-forme.

Dans la `build()`méthode du `FirstRoute`widget, mettez à jour le `onPressed()`callback :

```
      // Within the `FirstRoute` widget
      onPressed: () {
        Navigator.push(
          context,
          MaterialPageRoute(builder: (context) => const SecondRoute()),
        );
}
```

Accédez à la deuxième route à l'aide de Navigator.push()

Pour passer à une nouvelle route, utilisez la `Navigator.push()` méthode. La `push()`méthode ajoute un `Route`à la pile de routes gérées par le `Navigator`. D'où vient le `Route`? Vous pouvez créer le vôtre ou utiliser un `MaterialPageRoute`, ce qui est utile car il effectue la transition vers le nouvel itinéraire à l'aide d'une animation spécifique à la plate-forme.

Dans la `build()`méthode du `FirstRoute`widget, mettez à jour le `onPressed()`callback :

```
      // Within the `FirstRoute` widget
      onPressed: () {
        Navigator.push(
          context,
          MaterialPageRoute(builder: (context) => const SecondRoute()),
        );
}
```

Revenez au premier itinéraire en utilisant Navigator.pop()

Comment fermez-vous la deuxième route et revenez-vous à la première ? En utilisant la `Navigator.pop()`méthode. La `pop()`méthode supprime le courant `Route`de la pile de routes gérées par le `Navigator`.

Pour implémenter un retour à la route d'origine, mettez à jour le `onPressed()` rappel dans le `SecondRoute`widget :

```
      // Within the SecondRoute widget

      onPressed: () {

        Navigator.pop(context);

}
```

Code source

https://gist.github.com/NizarETH/fcffa380b8047fd60ccdb9ea93ecc505

- **Cordova/ionic**


APACHE
CORDOVA™
Your code
Your app
CORDOVA

Les applications que vous téléchargez sur vos smartphones ou tablettes peuvent être **conçues de différentes manières**.

Elles sont :

soit fabriquées pour être vues sur un seul type de système d'exploitation : une application téléchargée sur Android ne fonctionnera pas sur IOS. Ce sont là des applications dites natives. Il faudra les développer autant de fois que de systèmes d'exploitation différents.
soit, il existe des applications dites cross-platform (ou multiplateformes) ou d'autres dites hybrides qui sont compatibles avec tous les systèmes et supports.

Les deux ont leurs avantages et leurs inconvénients.

Cordova est un outil qui permet de créer des applications mobiles cross-platform.

Que permet-il de faire et quelles sont ses limites?

Un outil open-source performant:

Il existe plusieurs outils ou framework pour développer des applications mobiles. Cordova en est l'un d'eux. **Il utilise le langage javascript**. C'est une boîte à outils qui permet aux développeurs de travailler plus efficacement.

Vous le verrez souvent sous le nom de PhoneGap/Cordova ou Apache Callback.

Pourquoi ces deux noms?

En fait, PhoneGap a été développé par une société du nom de Nitobi Software qui a été racheté par Adobe. Ce dernier a transmis le projet à la Fondation Apache en 2011 qui le renomme Apache Cordova.

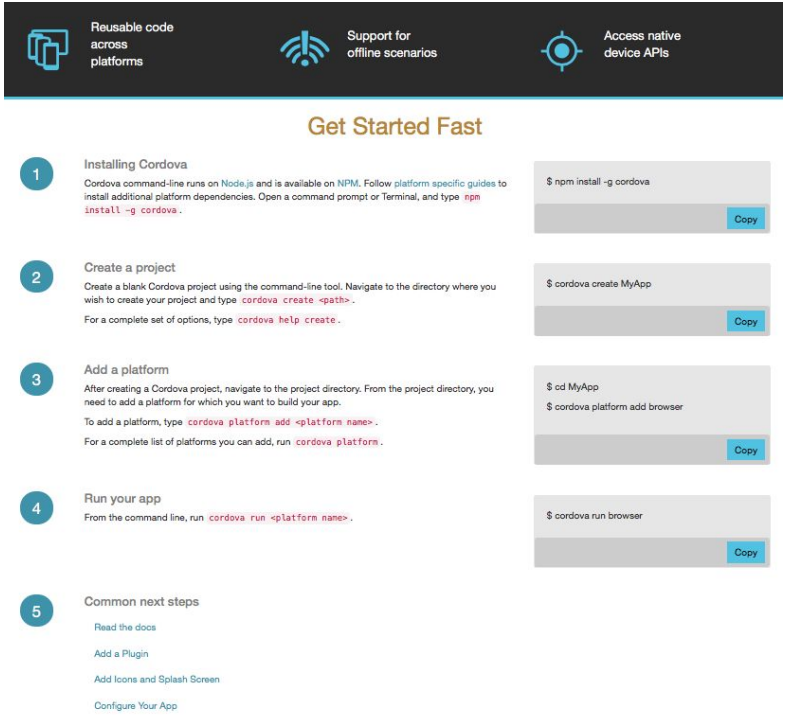
Pour information, la Fondation Apache est une organisation qui développe des logiciels Open Source. En gros, il s'agit d'une communauté de développeurs qui travaillent sur les mêmes projets de manière ouverte et décentralisée. Tous leurs logiciels sont en libre distribution. Leurs codes source sont disponibles pour tous. Il sont constamment enrichis, améliorés et testés par leurs utilisateurs.

Cordova est donc un outil open-source. Sa communauté d'utilisateurs est en constante progression.

Ses spécificités.

Il utilise les standards du web c'est-à-dire les technologies HTML, CSS et Javascript. En cela, les applications mobiles qu'il permet de concevoir sont dites cross-platform. Elles mélangent du web et des technologies natives qui permettent d'utiliser certaines fonctionnalités du smartphone.

Cordova **agit comme une surcouche** qui pourra permettre à l'application d'accéder et d'utiliser la caméra, la géolocalisation…Vous pourrez développer des interactions avec les bibliothèques d'images, les listes de contacts. Vous aurez la possibilité d'envoyer des notifications…


Reusable code across platforms
Support for offline scenarios
Access native device APIs
Get Started Fast
1
Installing Cordova
Cordova command-line runs on Node.js and is available on NPM. Follow platform specific guides to install additional platform dependencies. Open a command prompt or Terminal, and type npm install -g cordova.
$ npm install -g cordova
Copy
2
Create a project
Create a blank Cordova project using the command-line tool. Navigate to the directory where you wish to create your project and type cordova create <path>.
For a complete set of options, type cordova help create.
$ cordova create MyApp
Copy
3
Add a platform
After creating a Cordova project, navigate to the project directory. From the project directory, you need to add a platform for which you want to build your app.
To add a platform, type cordova platform add <platform name>.
For a complete list of platforms you can add, run cordova platform.
$ cd MyApp
$ cordova platform add browser
Copy
4
Run your app
From the command line, run cordova run <platform name>.
$ cordova run browser
Copy
5
Common next steps
Read the docs
Add a Plugin
Add Icons and Splash Screen
Configure Your App

Quelques limites

Il y a quelques soucis de performance dans cette solution. Ils ne sont pas relatifs à l'outil même mais au type d'application développée: le multi-plateforme.

> En effet, une application développée en cross-platform est idéalement une application relativement simple qui serait trop chère à développer en natif.
> Si vous avez besoin d'une application avec beaucoup d'animations par exemple il vaudra mieux opter pour du natif.
> Le design de ces applications est en général aussi moins poussé. L'expérience utilisateur sera peut-être moins pointue qu'avec une native. Mais il est toutefois possible de créer des interfaces efficaces.

L’installation d’un mode hors-ligne d’utilisation de l’application est aussi beaucoup plus difficile à concevoir sur du cross-platform.

De plus, les fonctionnalités de votre smartphone ne pourront pas être toutes exploitées au maximum comme avec une application native.

Par ailleurs, concernant Cordova même, ce n’est pas une solution prête à l’emploi. Son apprentissage prend du temps.

- **Xamarin**

Xamarin est bien gravé sur le marché, il n'a pas besoin d'introduction ou d'explication particulière. Néanmoins, il est impératif de dire qu'il s'agit de l'une des plateformes de programmation les plus populaires utilisées pour développer des applications mobiles fonctionnant à la fois sur des applications Android ou iOS. Les applications développées via C# et Net Framework sont créées dans Visual Studio et publiées sur diverses plateformes.
C'est un rêve devenu réalité pour les développeurs car s'ils développaient différentes applications pour différentes plates-formes, ils devraient y consacrer un nombre considérable d'heures et les entreprises qui les embaucheraient subiraient des pertes, en raison du temps, du paiement et des ressources du développeur. impliqué dedans. Xamarin réduit le temps et les efforts de développement, car il vous permet de créer des applications multiplateformes.

1) Framework équipé de Visual Studio

Comme il est équipé de Visual Studio, le framework vous permet de créer un IDE puissant et moderne. Il vous donne le framework .NET/C# nécessaire pour développer une sortie de code binaire réel natif performante afin de créer des applications natives hautes performances.

Vous pouvez également manipuler des fonctionnalités telles qu'un projet sophistiqué, une bibliothèque de modèles de projet, la complétion automatique du code et bien d'autres. Tout ce dont vous avez besoin, c'est de quelques semaines pour utiliser vos connaissances C# et la fonctionnalité de réutilisation du code de Xamarin pour publier une nouvelle application multiplateforme qui ressemble et se sent comme native.

2) Logique d’application partagée

Une autre caractéristique notable de la plate-forme est la logique de code partagé qui peut être écrite une seule fois, mais déployée sur différentes plates-formes. En effet, les composants tels que les objets métier, la logique d’application et les couches d’accès aux données sont tous partagés sur ces plates-formes.

Comme vous n’avez pas à écrire le code à partir de zéro, tout ce que vous avez à faire est de vous assurer d’écrire quelques codes uniques, et le reste sera partagé. Cela garantit que votre application atteint le marché en un rien de temps.

3) Interface utilisateur

Le développeur a la liberté de choisir la disposition de l'interface utilisateur, un grand avantage lors de la conception d'applications en fonction des intérêts des utilisateurs. Cependant, l'interface elle-même présente un inconvénient, dont nous parlerons plus tard.

4) Nuage de test Xamarin

Vous pouvez tester toutes vos applications rapidement dans le Xamarin Test Cloud et les amener à des niveaux inédits. Le cloud vous permet de tester n'importe quelle application sur des milliers d'appareils, et vous pouvez utiliser son incroyable système de reporting où vous pourrez identifier les goulots d'étranglement et les résoudre dans les plus brefs délais. Si vous recherchez la flexibilité, le cloud de test est la réponse.

5) Un incroyable magasin de composants

Le cadre est incroyable, vous disposez d'une incroyable collection de commandes d'interface utilisateur, de thèmes, de tableaux, de graphiques, de services cloud et d'une multitude de fonctionnalités puissantes pour vous aider à améliorer les fonctionnalités de votre application. Et cela contribue à la vitesse à laquelle vous pouvez développer votre application.

Inconvénients de Xamarin

Il y a quelques inconvénients à Xamarin, vérifiez-les ici

1) Frais généraux de l’application

La surcharge de l’application intégrée laisse une grande empreinte. Cela pourrait affecter le temps de téléchargement et l’espace de stockage requis pour l’application sur les appareils des utilisateurs. Bien sûr, l’équipe Xamarin fait de son mieux pour travailler sur ces problèmes, mais les utilisateurs de l’application doivent toujours en profiter au maximum.

2) Manque de soutien communautaire

C’était l’un des inconvénients les plus importants de Xamarin. Les développeurs Android, iOS et NET disposent d’ énormes forums, groupes et communautés de support communautaire avec une mine d’informations. Par rapport à tous ceux-ci, Xamarin est relativement nouveau, donc la communauté doit encore se développer. Si vous rencontrez des obstacles lors du développement de l’application, rien ne garantit que vous pourrez trouver une solution en ligne. Vous devrez peut-être embaucher un partenaire certifié Xamarin, un programme de partenariat qui vous aidera à résoudre vos problèmes car il a davantage accès aux ressources d’assistance.

3) Impossible de partager des codes en dehors de Xamarin

Les développeurs ne peuvent pas partager de codes avec d'autres développeurs ou équipes s'ils utilisent uniquement du code Swift, Java et Objective CA écrit à l'aide de C# et Xamarin, ne peuvent pas être partagés tels quels. veulent travailler sur ce cadre.

4) Certains codage manuel requis

L'expression scénario « écrivez un, codez n'importe où » n'est pas tout à fait vraie car certains des codes devront être écrits individuellement pour chaque plate-forme. Le développement de l'interface utilisateur pour chaque plate-forme prendra du temps, d'autant plus qu'il doit être effectué manuellement.

Conclusion

Xamarin est certainement un meilleur choix par rapport aux applications hybrides, car elles affichent simplement l'application dans un cadre sur l'appareil sur lequel vous l'exécutez et se connectent aux API spécifiques à la plate-forme pour la faire fonctionner. Cela le rend plus lent et moins élégant.

L'une des premières questions que les développeurs posent à leurs employeurs serait de savoir s'ils auraient besoin d'une application native ou d'une application multiplateforme. Si l'entreprise souhaite développer rapidement une application pour plusieurs plates-formes et offrir une sensation native, alors Xamarin serait un bon choix. L'embauche de professionnels hautement qualifiés faciliterait le processus pour vous.

C’est la fin du module

Merci !