Flutter - Dart

By Soultan HATSIJEV

ArtCoded - Intership

2022-06-27

Sommaire

Introduction

Flutter est un framework développé par Google pour permettre aux développeurs de créer des applications mobiles, desktop et web.

> *« Flutter est un kit de développement logiciel (SDK) d'interface utilisateur open-source créé par Google. Il est utilisé pour développer des applications pour Android, iOS, Linux, Mac, Windows, Google Fuchsia et le web à partir d'une seule base de code. » (Wikipedia : Flutter)*

Pour développer ces fameuses applications, le programmeur utilisera le langage *Dart*, ce dernier est également développé par Google, c'est un langage compilé qui se base sur la syntaxe de C++. Il est aussi composé d'un *Garbage Collector*[1].

Ce framework permet donc à un/des développeur(s) d'optimiser leurs temps de travail en se concentrant sur une application qui sera ensuite partagée sur les différents plateformes prévues.

Évidemment, expliquer Flutter de cette manière sera très simple et basique, pour bien comprendre son fonctionnement, il nous faut d'abord comprendre le langage utilisé : Dart.

1 Pour plus d'information concernant le fonctionnement des *Garbage Collector* (Collecteur de miettes), suivez le lien suivant « Garbage collection fundamentals »

Dart

Avantages

Google a donné un second souffle à Dart, ce dernier a, pour l'occasion, retrouvé des couleurs, en effet, Google a préféré partir sur son utilisation c'est principalement grâce à ses modes de fonctionnement :

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

Évidemment, ces deux compilateurs ne sont pas les seuls avantages de Dart :

- Multiplateforme
- Langage typé
- Supporté par Google

Ahead Of Time

Ce mode de fonctionnement permet de générer une application native pour chaque plateforme. Cette méthode est économe en coût.

En effet, lors du développement d'application mobile, il est aujourd'hui impossible de se passer d'une des deux plateformes dominantes (iOS, Android), et pour justement développer des applications sur chaque plateformes de manière native, il faut soit passer par Swift ou ObjectiveC pour iOS ou par Kotlin ou Java pour Android. Ainsi, pour palier à ce problème, il existe différents frameworks tel que Flutter dans notre cas, mais il existe aussi *React Native* dont le code est principalement composé de JavaScript et de web classique (HTML CSS) ou encore *Xamarin*, prochainement *MAUI*, pour Microsoft dont les langages sont principalement C# pour le backend et XAML pour le frontend.

Sauf que contrairement à ses deux concurrents, Dart permet aussi de compilé son code pour les plateformes *desktop* macOSX, Linux et enfin Windows mais aussi pour le web en JavaScript.

Just-In-Time

Le second fonctionnement intéressant de Dart est le JIT, ce dernier va compilé le code dans le langage natif à la plateforme juste avant l'exécution du programme. Cette fonction est évidemment aussi disponible avec Java notamment.

Le compilateur JIT va tenter de prédire quelle instruction suivante va être exécuter afin de compiler à l'avance.

Enfin, ce fonctionnement permet le *Hot Reload*, rechargement à chaud en français, qui permet au développeur de visualiser les changements apportés au code sans devoir éteindre et recompiler toute l'application, il n'y a que les passerelles modifiées qui seront recompilées.

Le compilateur JIT fait partie de la machine virtuelle de Dart, il supporte l'interprétation pure du code et l'optimisation de l'exécution.

Multiplateforme

Un des gros avantages de Dart est sa compatibilité avec toutes les plateformes. Un développeur peut développer une application *desktop* pour toutes les plateformes mais aussi depuis n'importe quelle plateforme, contrairement à *WPF*, le framework d'application *desktop* limité à Windows.

Langage typé

Contrairement à des langages comme Python ou Javascript, Dart est typé de manière optionnelle, c'est-à-dire que l'on peut utiliser les types mais ils ne sont pas obligatoires, ces derniers peuvent en effet être déterminé par le compilateur lors de l'exécution.

```
// Explicit type definition
Map<String, dynamic> arguments = {'argA': 'hello', 'argB': 42};

// Let Dart infer the type (final can also be used instead of var)
var arguments = {'argA': 'hello', 'argB': 42}; // Map<String, Object>
```

Supporté par Google

Enfin, Dart étant un langage crée par Google, il est aussi supporté par ces derniers. Une telle organisation derrière un projet confirme évidemment son sérieux mais aussi sa maintenance et sa viabilité.

De plus, on peut être sûre que le projet aura une très bonne documentation, régulièrement mis à jours

Inconvénients

Nous avons listés les avantages majeurs à l'utilisation de Dart, évidemment, qui dit avantages dit aussi inconvénients. La liste des inconvénients est clairement moins longue que son homonyme mais il n'empêche que le langage en possède quelques-un :

- Son manque de maturité
- Limité en terme d'outils et de librairies
- Opérabilité sur iOS

Le manque de maturité

Dart en lui-même n'est pas tout jeune, il est apparu en 2011, par contre le langage a subit plusieurs mutation tout au long de son histoire (Wikipedia : Dart #Histoire). Le manque de maturité correspondant plutôt à Flutter qui n'est sortis qu'en 2017, ce qui est assez jeune dans le monde de l'informatique.

Évidemment lorsqu'une entité comme Google se trouve derrière le projet, on peut être assuré de son sérieux, cependant cela n'empêchera quand même pas la technologie d'avoir des gros manques sur certains aspects que seuls les programmeurs découvriront au fur et à mesure de leur aventure.

Limité en terme d’outils et de librairies

Cet inconvénient rejoint le précédent, la technologie étant assez nouvelle, elle n’est pas aussi fournie que NodeJs, C# ou en Java en terme de *package* développés par la communauté. Il faudra donc que le développeur s’occupe de développer lui-même certaines fonctionnalités qu’il ne pourra pas trouver sur internet.

Opérabilité sur iOS

Ce dernier point est peut-être obsolète, en effet, lors des premières versions de Flutter, certaines fonctionnalités propre à iOS était soit bugué soit simplement indisponible. Des fonctionnalités tels que *VoiceOver*, *Guided Access*, *Captioning*, etc n’étaient pas implémentés ou bien lorsqu’une application Flutter prenait une photo sur iOS, cette dernière perdait toutes ses données *EXIF*[2].

Évidemment, vu l’évolutivité du framework, on peut parier que Google a réglé ou réglera ces bugs très prochainement, un test sérieux nous le dira, cependant le monde professionnel ne compte pas les attendre.

Breaking changes

Un dernier inconvénient remarqué principalement sur les forums de discussions concerne les mises à jours. Plusieurs dizaines d’utilisateurs, si ce n’est plus au vu des likes, se plaignent des changements majeurs qui rendent obsolète un ancien code Flutter et ce en moins d’un an.

Évidemment, la plus part de ces post étaient écris en 2020, 2021, nous savons que dans le monde de l’informatique, les choses changent très vite, il faut donc prendre cela avec des pincettes.

Comparaison

Étant donné que Dart est un langage compilé et non pas interprété, nous allons le comparer à deux autres langages qui offrent les mêmes (à quelques détails près) fonctionnalités, c’est-à-dire C# et Java.

Popularité

Tout d’abord, nous allons comparer C#, Java et Dart sur base des critères suivants :

- Les *Pull Request* ouvertes
- Les *Push* effectués
- Les *Issues* ouvertes
- Les projets *Stars*
- Nombre de questions sur StackOverflow
- Les informations sur GitHub
- Les offres d’emplois sur ICT Jobs

2 Les données EXIF sont les données de localisation, d’orientation, de date, de gamma d’une photo.

Pull Requests[3]

ANNÉE	2020		2021		2022
TRIMESTRE	**T1**	**T4**	**T1**	**T4**	**T1**
C#	3,84	3,67	3,61	3,37	3,06
Java	10,94	11,55	11,68	12,21	13,08
Dart	0,48	1,08	1,18	0,69	0,79

Push

ANNÉE	2020		2021		2022
TRIMESTRE	**T1**	**T4**	**T1**	**T4**	**T1**
C#	4,84	2,79	2,42	1,08	64,05
Java	11,75	9,12	8,6	3,65	4,21
Dart	0,3	0,34	0,34	0,14	0,17

Issues

ANNÉE	2020		2021		2022
TRIMESTRE	**T1**	**T4**	**T1**	**T4**	**T1**
C#	5,68	5,71	5,65	5,2	4,71
Java	11,48	11,31	11,79	11,14	10,89
Dart	2,17	2,16	2,39	2,1	2,07

Stars

ANNÉE	2020		2021		2022
TRIMESTRE	**T1**	**T4**	**T1**	**T4**	**T1**
C#	3,54	3,48	3,74	3,83	3,87
Java	9,79	9,67	9,28	8,58	8,39
Dart	0,74	0,86	0,84	0,59	0,68

3 Les pourcentages représentes les « parts de marché » qu'occupe chacun des langages sur la plateformes GitHub. Toutes les statistiques ont étés récupérées sur le site suivant « GitHut 2.0 »

StackOverflow

	ACTIVE	UNANSWERED[4]	RATIO[5]
C#	1 544 867	406 441	26,31
Java	1 854 086	537 530	28,99
Dart	72 030	25 897	35,95

GitHub

	PLUS POPULAIRE[6]	STARS[7]	CONT.[8]	ISSUES[9]	PR	PUB. REP.[10]
C#	dotnet/aspnetcore	28,8	1 049	2 314	70	39 520
Java	openjdk/jdk	13,4	608	/	182	137 586
Dart	dart-lang/sdk	8,2	396	6 580	3	16 378

ICT Jobs[11]

- **.NET[12] :** 114 offres
- **Java[13] :** 94 Offres
- **Dart[14] :** 0 Offre

Conclusion

On peut voir sur chacun des tableaux que Dart prend de l'engouement au 4ème trimestre de 2020 (T4 - 2020) et le 1er trimestre de 2021 (T1 -2021) ce qui correspond à l'annonce et le lancement de Flutter 2 qui permettait d'enfin développer des applications de bureaux Linux, Windows et Mac.

Lorsqu'on observe les chiffres des T2 et T3 pour la catégorie ***Stars*** par exemple toujours sur GitHut, l'engouement est toujours là, jusqu'au 4ème trimestre de 2021 où la chute est considérable.

Les trois langages sont assez stables dans l'ensemble, sauf quelques exceptions (***Push*** C# : T1 – 2022).

---

4 *Issue* dont aucune réponse n'a été voté comme la réponse correcte ou dont aucune réponse n'a été acceptée comme réponse correcte

5 Représente le pourcentage de questions sans réponses suivant la formule suivante : Ratio = (nombre d'*unanswered questions* / nombre d'*active questions*) * 100

6 Le projet le plus populaire est le *repository* ayant eu le plus de *stars* sur la page GitHub du langage

7 Le nombre de *stars* est compté en millier

8 Représente le nombre de contributeurs

9 Représente le nombre d'*issues* ouvertes

10 Représente le nombre de *repository* publiques

11 La recherche est faites en sélectionnant la technologie dans la liste des technologies disponibles

12 La recherche .NET comprend : *.NET Core, ASP.NET, C#, VB.NET, Xamarin*

13 La recherche Java comprend : *Grails / Groovy, GWT, JBoss (WildFly) / Tomcat, Jenkins, JPA / Hibernate, JSF, JSP / Servlets, JUnit, Maven, Spring, Struts*

14 Dart n'étant pas dispnible dans la liste des technologies, la recherche a été faites sur base des mots-clés suivant : *Dart* et *Flutter*

Ensuite, sur *StackOverflow*, le nombre de questions en rapport avec Dart est très minime comparé à ses compères alors qu'elle a un ratio de questions restées sans réponses d'environ 7 % plus élevé que les deux autres.

Les statistiques sur les projets les plus populaires pour chacune des technologies sur GitHub montre un écart de popularité encore énorme entre les deux mastodontes et notre petit poucet. On peut voir par contre que le nombre de projets publiques pour Dart n'est que de moitié moins que pour C#.

Enfin, la statistiques (limitées dans ce cas évidemment) la plus intéressante, l'utilisation dans un contexte professionnel. En Belgique, ICT Jobs est un site de référence pour la recherche d'emploi dans l'informatique et on se retrouve donc avec 0 offre pour Flutter et Dart, ce qui est décevant.

Organisation

En C# l'organisation des classes fonctionne avec les *namespace*, le langage ne prend pas en compte la localisation du fichier dans le système, il se repère via le *namespace*.

Java est similaire à C# avec son système de *package*.

En Dart, il est possible de fonctionner aussi bien en package qu'en fichier tel qu'en JavaScript lors de l'appelle d'un module ou d'un fichier.

Ainsi en C#, si nous tentons de faire communiquer une classe Voiture se trouvant dans le *namespace Models* et une classe VoitureService se trouvant dans le *namespace Services*, notre code ressemblera à cela

```
namespace MonApp.Services
{
	public class VoitureService
	{
		public VoitureService()
		{
			// Logic du constructeur
		}
	}
}

using MonApp.Services ;

namespace MonApp.Models
{
	public class Voiture
	{
		publlic Voiture()
		{
			VoitureService voitureService = new VoitureService ;
		}
	}
}
```

Dans la classe *Voiture*, pour accéder au service *VoitureService*, je déclare un *using* du *namespace* dans lequel il se trouve.

Dans un exemple similaire en Dart, cela nous donnerait un code dans ce style :

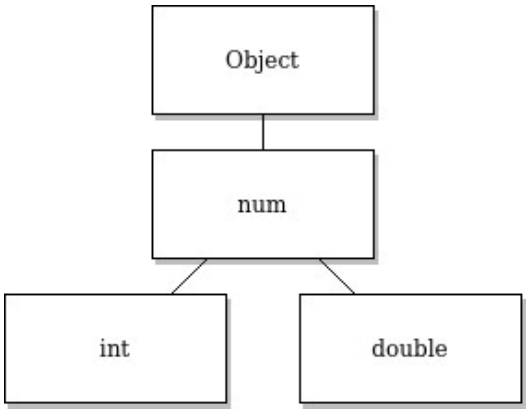
```dart
// Fichier nommé Voiture.dart

int nbVoiture = 5 ; // Variable Top Level

class Voiture {
        // Logique de la classe Voiture
}
// Fichier nommé Main.dart

import 'Voiture.dart' ;

// Il est aussi possible de nommé le fichier pour notamment récupérer des variables Top Level

import 'Voiture.dart' as Car ;

// Il est aussi possible d'importer le package Voiture qui représente en faite une organisation de
// dossier

//   voitures/
//      pubspec.yaml— Défini le nom du package, les dépendant, etc.
//      lib/
//         Voiture.dart— Import et export, c'est le fichier qui est importé
//         src/
//            Mercedes.dart— Autre code
//      bin/
//         Conduire.dart— Contient la fonction main() qui est le point d'entré de l'application (si c'est
//                        un package exécutable ou s'il contient des outils exécutable)

import 'package:Voiture' ;


main() {
        Voiture voiture1 = new Voiture() ;
        // ou bien
        var voiture2 = new Voiture() ;

        print(Car.nbVoiture) ;
}
```

Les types

Concernant le typage des données, Dart le gère évidemment, cependant le types ne sont pas exactement les mêmes qu'en C# ou Java. Dans ces derniers, pour les entiers par exemple, il est existe plusieurs types tels que les *short* qui sont des entiers de 16 bits, les *int* qui sont de 32 bits et enfin les *long* qui sont des entiers de 64 bits.

Dans notre cas, le tableau suivant recense les différents types :

Type	Explication
int	Représente un nombre entier, il n'y a pas de différence en Dart concernant la taille en bits, dans ce cas, c'est du 64 bits
double	Représente un nombre réel de 64 bits
num	*num* est en fait l'objet qu'hérite les *int* et les *double*, num peut donc être utilisé pour les deux types de données
string	Représente une chaîne de caractères encodé en UTF-16
bool	Représente un booléen et ne peut prendre que *true* ou *false* (vrai ou faux)
List<E>	Dart ne contient pas de tableau à taille fixe, les *array* seront donc toujours des listes, cependant, il est possible de les instancier de deux manières différentes, une donnant une liste à taille fixe, l'équivalent d'un *array* ou à taille variable. Nous verrons cela le code juste après.
Map<Tkey, Tobject>	Les *Map* en Dart sont l'équivalent des *Dictionnary* en C# ou des *HashMap* en Java



Représentation du fonctionnement des types *int* et *double* en Dart. Ces deux types héritent de *num* qui lui-même hérite d'*Object*.

Il est aussi possible d'utiliser le mot clé *dynamic* dans une liste ou un dictionnaire notamment (*List<dynamic>* ou *Map<string, dynamic>*) pour que les objets qui seront ajoutés à la liste n'aient pas tous la même valeurs. C'est une fonctionnalité aussi disponible en C#, **CEPENDANT** il est déconseillé de travailler de cette manière, l'utilisation d'un *dynamic* risque d'embrouiller le développeur. Ce n'est pas pour rien que des types sont disponibles, il est donc conseillé de les utiliser.

Nous avons vu dans le tableau qu'il existait deux manières d'instancier une liste. Nous verrons ces manières maintenant.

Les listes fixes (Dart : List<E>)

```
final fixedLengthList = List<int>.filled(5, 0); // Creates fixed-length list.

print(fixedLengthList); // [0, 0, 0, 0, 0]

fixedLengthList[0] = 87;
fixedLengthList.setAll(1, [1, 2, 3]);

print(fixedLengthList); // [87, 1, 2, 3, 0]

// Fixed length list length can't be changed or increased

fixedLengthList.length = 0;  // Throws
fixedLengthList.add(499);    // Throws
```

Les listes à tailles variables (Dart : List<E>)

```
final growableList = <String>['A', 'B']; // Creates growable list.

growableList[0] = 'G';

print(growableList); // [G, B]

growableList.add('X');
growableList.addAll({'C', 'B'});

print(growableList); // [G, B, X, C, B]
```

Dynamic

Il peut y avoir des cas où un développeur voudrait créer une liste contenant des objets différents par exemple des *Mercedes* et des *BMW* dans une seule liste de voiture. Pour ce faire, un bon développeur n'utilisera pas le *dynamic* mais se basera plutôt sur le développement orienté objet et principalement le concept d'héritage.

Donc si nous voulons créer une liste contenant des voitures différentes, nous créerons d'abord une classe parent *Car*, dont hériteront tous les modèles de voitures restants.

```
abstract class Car {
	String reference ;
	int horsePower ;
	DateTime releaseDate ;

	Car(String reference, int horsePower, DateTime releaseDate) {
		this.reference = reference ;
		this.horsePower = horsePower ;
		this.releaseDate = releaseDate ;
	}

	// Les différents méthodes de getter et setter
}


class Mercedes extends Car {

	Mercedes(String reference, int horsePower, DateTime releaseDate)
	 : super(reference, horsePower, releaseDate) {
		// Logique du constructeur
	}
}

main() {

	Mercedes mercedes = new Mercedes("W204", 136, DateTime.utc(2014,06,01)) ;
	BMW bmw = new BMW("E46", 152, DateTime.utc(2004,02,01)) ;

	final cars = new List<Car>[mercedes, bmw]; // growing list
}
```

Les boucles

Un autre sujet primordial dans le monde de la programmation reste les boucles. Ces dernières d'un langage à l'autre ne changent qu'à quelques détails près.

Nous allons définir un exemple en C# des trois boucles *while*, *for* et *foreach* :

```
public void Loops()
{
	int[] numbers = new int[] { 1, 2, 3, 5, 6, 7, 8 };

	int k = 0 ;

	while(k < numbers.Length)
	{
		Console.WriteLine(numbers[k]) ;
		k++ ;
	}
	// Afichera 1, 2, 3, 4, …, 8

	for(int i = 0; i < numbers.Length; i++)
	{
		Console.WriteLine(numbers[i]);
	}
	// Affichera 1, 2, 3, …, 8

	foreach(int j in numbers)
	{
		Console.WriteLine(j);
	}
	// Affichera 1, 2, 3, …, 8
}
```

Je n'expliquerai pas le principe de ces trois boucles, elles sont connus et internet est notre ami.

En Dart, ces boucles sont exactement similaires en tout point à C#. La seule chose qui change est l'initialisation du tableau.

```
List<int> numbers = [1,2,3,4,5,6,7,8];

 int k = 0;
 print("While loop");
 while(k < numbers.length) {
  print(numbers[k]);
  k++;
 }

 print("For loop");
 for(int i = 0; i < numbers.length; i++) {
  print(numbers[i]);
 }

 print("Foreach loop");
 for(int j in numbers) {
  print(j);
 }
```

Il existe cependant quelque chose en plus en Dart, quelque chose de similaire à JavaScript, c'est la fonction *forEach(void action(E element))* (Dart: forEach method) qui se présente de cette manière :

```
void main() {
	List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8] ;

	numbers.forEach(print); // Affichera 1, 2, 3, …, 8

	// le paramètre de cette fonction représente une méthode, nous pouvons donc également
	// écrire quelque chose dans ce style

	numbers.forEach((element) => {
		if(element % 2 == 0) {
			print("even")
		} else {
			print("odd")
		}
	});
}
```

Les objets

Il faut comprendre en Dart que tout est un objet, comme nous l'avons vu précédemment, même les types (*int, double, bool*, etc) le sont.

Pour déclarer un objet en Dart, c'est l'équivalent de presque tous les langages de programmation, une création de classe avec un constructeur, des membres public, privé, protégé, des accesseurs et des mutateurs, etc.

Il n'y a pas de mot clé *private*, *public*, *protected* pour rendre un membre privé en Dart, il suffit d'écrire le nombre de la variable en commençant par un *underscore* : « _ ».

Le fonctionnement des membres privés est assez étrange, en effet si une classe se trouve dans le même package qu'une autre, elles peuvent toutes les deux accéder au membres privés de l'un et de l'autre, cependant si nous créons une classe dans un autre package, cette dernière n'aura pas accès aux membres privés.

Depuis récemment, il est possible d'utiliser des annotations pour par exemple rendre une méthode protégé (*protected*) ou encore dépréciée (*deprecated*) et ceux de la manière suivante :

```
class Voiture {

	String modele ;
	String _numeroChassis ;

	// Constructeur
	Voiture(String modele, String numeroChassis) {
		// Logique du constructeur
		this.modele = modele;
		this._numeroChassis = numeroChassis;
	}

	// Accesseur pour la propriété _numeroChassis
	String RecupererNumeroChassis() {
		return _numeroChassis ;
	}

	// Membre protégé
	@protected
	void Demarrer() {
		// Logique
	}

	// Membre déprécié
	@deprecated
	void Accelerer() {
		// Logique
	}
}

void main() {
	Voiture v = new Voiture("E46", "UF-36552");
	v._numeroChassis = "UF-0000"; // Ceci fonctionne, nous sommes dans le même package
}
```

Notre classe Voiture est maintenant prête, nous allons l'appeler depuis le package principal *Main*.dart

```
import 'Voiture.dart' ;

void main() {
	Voiture mercedes = new Voiture("W204", "UT-0000");
	mercedes._numeroChassis = "UF-36552"; // Retourne une erreur, le membre est
	                                      // inaccessible
	print(mercedes.RecupererNumeroChassis()); // Affiche "UT-0000"
	mercedes.modele = "W210";
	print(mercedes.modele); // Affiche W210 au lieu de W204 (propriété public)

	mercedes.Accelerer(); // Affichera un avertissement disant que la classe est dépréciée
}
```

Si maintenant nous voulons créer une classe qui héritera de Mercedes

```
import ‘Voiture.dart’;

class Mercedes extends Voiture{

	// Constructeur
	Mercedes(String modele, String numeroChassis)
	 : super(modele, numeroChassis) {
		// Logique du constructeur
	}

	@override
	void Demarrer() {
		// Nouvelle logique propre à la classe AMG
	}
}
```

Abstract, Interface et *Mixin*

Interfaces

En Dart, il n’existe pas d’*Interface*, toutes les classes implémentes implicitement une interface, en d’autre termes toutes les classes peuvent être utilisées comme des interfaces.

```
class IVoiture {
	void demarrer();
	void freiner();
	void verrouiller();
	void deverouiller();
}
```

La classe *IVoiture* nous servira d’interface, nous pouvons maintenant l’implémenter.

```
import ‘IVoiture.dart’;

class Mercedes implements IVoiture {
	Mercedes() {

	}

	void demarrer() {
		// instructions
	}

	void freiner() {
		// instructions
	}

	// etc pour toutes les méthodes
}
```

Classe abstraite ou *Abstract*

Le mot clé *abstract* existe bel et bien, nous pouvons créer (vu précédemment) des classes abstraites, ces classes ne peuvent pas être instanciées mais peuvent être implémentées ou encore héritées.

Il est plus intéressant d'utiliser des classes abstraite en guises d'interface, d'un point de vue organisationnel ou encore structurel, le code rendra mieux et sera mieux définis.

Mixin

'*Les mixins sont un moyen de réutiliser le code d'une classe dans plusieurs hiérarchies de classes.*' (Dart : Language tour)

Les *mixins* ne contenant pas de constructeur, il n'est pas possible de les instancier, elles pourraient s'apparenter à des interfaces dont les fonctions sont implémentées.

Si nous reprenons les exemples sur la documentation de Dart, nous aurons ceci :

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

Il n'est pas possible d'hériter ou d'implémenter un *mixin*, pour qu'une classe puisse l'utiliser, ce dernière doit hérité d'un classe sur laquelle le *mixin* sera ajouté, de la manière suivante :

```
class Musician {
  // ...
}
mixin MusicalPerformer on Musician {
  // ...
}
class SingerDancer extends Musician with MusicalPerformer {
  // ...
}
```

Asynchrone

La programmation asynchrone est devenu extrêmement important dans le monde de la programmation, si celle-ci est indisponible, le langage ne perdurera pas.

Il est possible en Dart de travailler de manière asynchrone, cependant, cette manière de travailler diffère fortement de C# ou Java.

Prenons le code suivant en C#. Une fonction asynchrone qui retourne un booléen et une fonction synchrone qui retourne également un booléen. La seconde fonction appelle la première de manière synchrone.

```
public async Task<bool> DoSomethingAsync()
{
        await Task.Delay(10000) ;
        return true;
}

public bool DoSomething()
{
        return Task.Run(async () => await DoSomethingAsync()).Result;
}
```

Travailler de cette manière est simplement impossible en Dart. En C#, il est possible d'attendre le résultat (la valeur de retour) d'une tâche, or en Dart, pour qu'une méthode asynchrone soit appelée dans une méthode synchrone, elle ne doit pas retourner de valeur, elle doit être *void*. Comme dans l'exemple suivant :

```
Future<bool> doSomethingAsync() async {
  Future.delayed(const Duration(seconds: 10));
  return true;
}

void doSomething() {
  doSomethingAsync()
    .whenComplete(() {
      print("doSomethingAsync completed!");
    });
}
```

Isolates

En Dart, le code est exécuté dans un *isolate* (isolat) et chacun n'a qu'un seul thread d'exécution dans lequel il ne partage absolument aucun objet mutable avec d'autre isolats, en d'autres termes, il ne partage pas sa mémoire avec d'autres isolats. Pour qu'ils puissent communiquer, ils vont se transmettre des messages.

Cependant, depuis Dart 2, la plateforme web ne prend plus en charge les isolats et il est donc suggéré aux développeurs d'utiliser les *Web Workers* à la place.

Il est possible de faire communiquer deux isolats de la manière suivante (Code repris à la source suivante (Dart: 2-Way Communication)):

```dart
// Example of bi-directional communication between a main thread and isolate.

import 'dart:io'; // for exit();
import 'dart:async';
import 'dart:isolate';

Future<SendPort> initIsolate() async {
  Completer completer = new Completer<SendPort>();
  ReceivePort isolateToMainStream = ReceivePort();

  isolateToMainStream.listen((data) {
    if (data is SendPort) {
      SendPort mainToIsolateStream = data;
      completer.complete(mainToIsolateStream);
    } else {
      print('[isolateToMainStream] $data');
    }
  });

  Isolate myIsolateInstance = await Isolate.spawn(myIsolate, isolateToMainStream.sendPort);
  return completer.future;
}

void myIsolate(SendPort isolateToMainStream) {
  ReceivePort mainToIsolateStream = ReceivePort();
  isolateToMainStream.send(mainToIsolateStream.sendPort);

  mainToIsolateStream.listen((data) {
    print('[mainToIsolateStream] $data');
    exit(0);
  });

  isolateToMainStream.send('This is from myIsolate()');
}

void main() async {
  SendPort mainToIsolateStream = await initIsolate();
  mainToIsolateStream.send('This is from main()');
}
```

Flutter

On a beaucoup parlé de Dart, sa syntaxe et le code, mais Flutter dans tout ça ? Flutter utilise Dart comme langage de programmation, la création de *widget*, de page, de fenêtre, tout se fait en Dart en utilisant évidemment les API mises à disposition par Google.

Lorsqu'on crée un projet *template* de Flutter, nous avons un fichier « *main.dart* » qui contient tout le code.

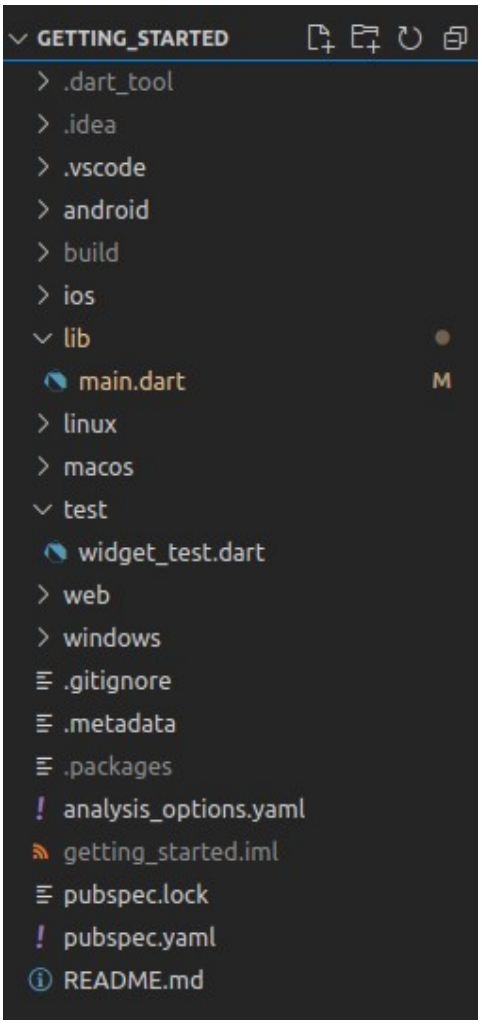
Fichier « *main.dart* »

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

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

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

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

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

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

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

  void _incrementCounter() {
    setState(() {
      // This call to setState tells the Flutter framework that something has
      // changed in this State, which causes it to rerun the build method below
      // so that the display can reflect the updated values. If we changed
      // _counter without calling setState(), then the build method would not be
      // called again, and so nothing would appear to happen.
      _counter += 2;
    });
  }
```

```dart
  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called, for instance as done
    // by the _incrementCounter method above.
    //
    // The Flutter framework has been optimized to make rerunning build methods
    // fast, so that you can just rebuild anything that needs updating rather
    // than having to individually change instances of widgets.
    return Scaffold(
      appBar: AppBar(
        // Here we take the value from the MyHomePage object that was created by
        // the App.build method, and use it to set our appbar title.
        title: Text(widget.title),
      ),
      body: Center(
        // Center is a layout widget. It takes a single child and positions it
        // in the middle of the parent.
        child: Column(
          // Column is also a layout widget. It takes a list of children and
          // arranges them vertically. By default, it sizes itself to fit its
          // children horizontally, and tries to be as tall as its parent.
          //
          // Invoke "debug painting" (press "p" in the console, choose the
          // "Toggle Debug Paint" action from the Flutter Inspector in Android
          // Studio, or the "Toggle Debug Paint" command in Visual Studio Code)
          // to see the wireframe for each widget.
          //
          // Column has various properties to control how it sizes itself and
          // how it positions its children. Here we use mainAxisAlignment to
          // center the children vertically; the main axis here is the vertical
          // axis because Columns are vertical (the cross axis would be
          // horizontal).
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ), // This trailing comma makes auto-formatting nicer for build methods.
    );
  }
}
```

Organisation du projet



Le code de notre projet se trouvera dans le dossier *« lib »*, ce dernier contiendra très probablement d'autre dossiers et ainsi de suite pour garder une architecture propre et soignée.

On peut apercevoir un dossier *« test »*, il contient les *Unit Tests* du projet.

Ensuite plusieurs dossiers en fonction du système d'exploitation utilisé (Windows, macOS, Linux, Android, iOS), ceux-là contiennent les fichiers nécessaires au déploiement sur la bonne plateforme. Ils sont crées ou modifier lors de la compilation du projet Flutter.

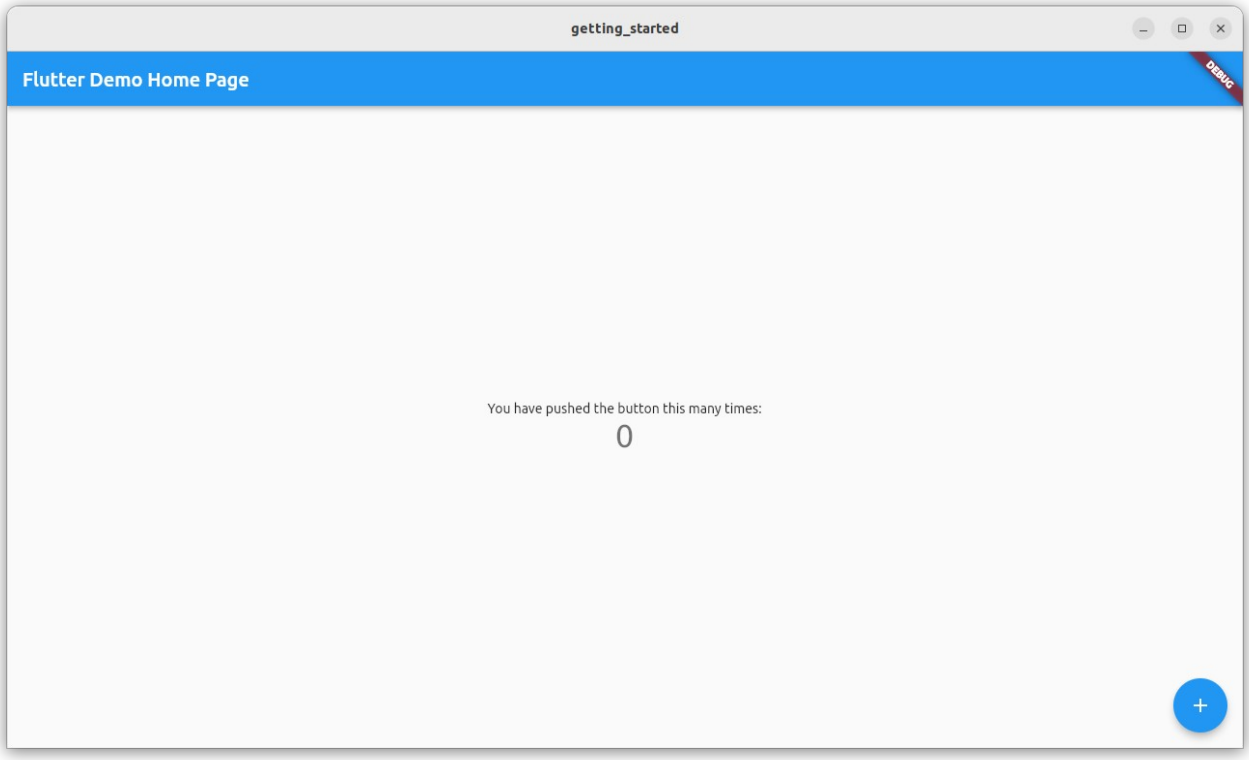
Plusieurs architecture sont possibles avec Flutter, mais les plus intéressantes reste MVVM (Model, View, ViewModel)[15] et BLoC (Business Logic Component)[16].

Nous ne nous intéresserons pas aux autres fichiers.

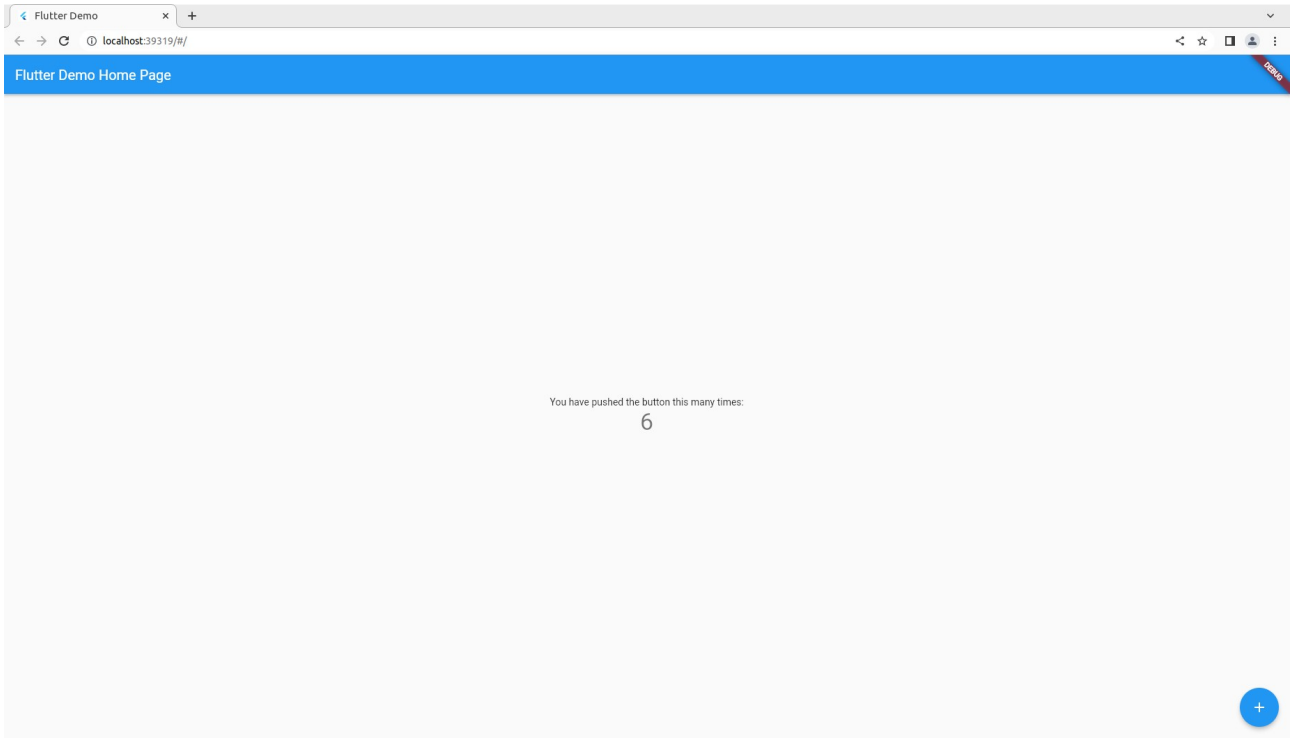
15 Démonstration disponible au lien suivant « Flutter: MVVM Architecture »
16 Explication disponible au lien suivant « How to Implement the BLoC Architecture in Flutter: Benefits and Best Practices »

Démonstration

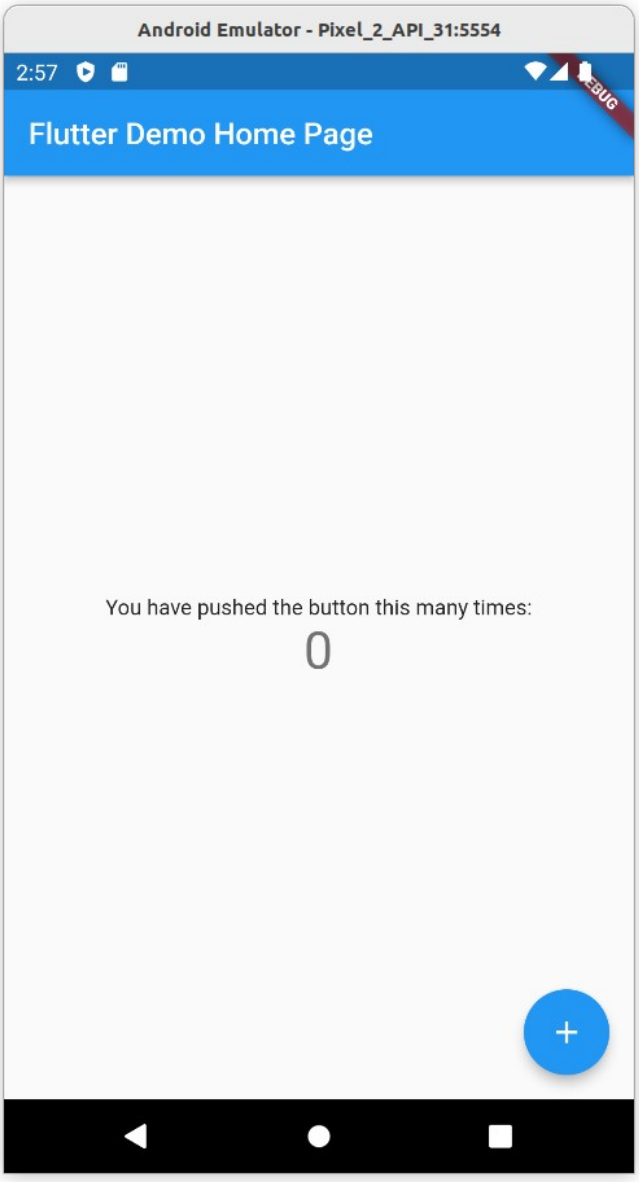
Flutter Linux



Flutter Web



Flutter Android



Alors ?

Avec cette unique application *template*, nous nous retrouvons avec une application cross-plateformes, fonctionnant aussi bien sur Linux, sur Android et sur Chrome en web.

Là intervient l'avantage mentionné plus tôt « *Multiplateforme* », le gain de temps et évidemment d'argent que permet cette technologie n'est pas des moindre.

Cependant, nous avons lancé ici une application extrêmement basique, sans aucune réelle logique poussée. Nous apercevrons les limites de Flutter lors du développement en situation professionnelle.

Conclusion

Au terme de ce rapport, on peut féliciter Google, Flutter est une technologie intéressante qui a redonné du souffle à Dart qui lui-aussi est un langage intéressante et ludique à l'apprentissage.

Malgré que les avantages surpassait les inconvénients, il y a un point majeur qui m'inquiète toujours : l'évolutivité.

Flutter est jeune, Dart également, Google travail intensément sur leur développement mais ceci comprend aussi des modifications majeures qui risquent de vite rendre la technologie pesante. Le fait que la plus part des sujets des forums critiquant Flutter datent d'un an ou plus me rend évidemment optimiste mais seule une utilisation continue me permettra de voir si réellement ça à changé, ou si c'est bien comme les critiques le disent.

Concernant Dart cependant, le langage est assez facile à prendre en main, la documentation est non seulement complète mais aussi détaillée sur les points importants et surtout, il existe une communauté active sur internet. Étant un habitué de C#, la transition ne sera pas compliquée, il en va de même pour les développeurs Java qui se retrouveront très facilement dans la manière d'écrire le code.

Pour conclure, je peux dire que Flutter est réellement un atout à prendre dans notre aventure en tant que développeur et quitte à ce que la technologie ne décolle jamais, il est toujours bien d'avoir pu s'essayer à quelque chose mélangeant correctement innovation et tradition.

Références

Wikipedia : Flutter: , Flutter (logiciel), , https://fr.wikipedia.org/wiki/Flutter_(logiciel)
Wikipedia : Dart #Histoire: , Dart (langage), , https://fr.wikipedia.org/wiki/Dart_(langage)#Histoire
Dart : List<E>: , List<E> class, , https://api.dart.dev/stable/2.16.2/dart-core/List-class.html
Dart: forEach method: , forEach method , , https://api.dart.dev/stable/2.17.5/dart-core/Iterable/forEach.html
Dart : Language tour: , A tour of the Dart language, , https://dart.dev/guides/language/language-tour#adding-features-to-a-class-mixins
Dart: 2-Way Communication: Leland Zach, Dart Isolate 2-Way Communication, 2019, https://medium.com/@lelandzach/dart-isolate-2-way-communication-89e75d973f34