



# Formation Flutter

Kinable Antoine - Boustani Mehdi - 16 octobre 2024



**N-HiTec**

Allée de la Découverte 10, 4000 Liège, Belgium

[nhitec.com](https://nhitec.com) | [info@nhitec.com](mailto:info@nhitec.com)

# Table des matières

<b>Table des matières</b>	<b>2</b>
<b>I Introduction</b>	<b>4</b>
1 Prélude . . . . .	4
2 Prérequis . . . . .	4
3 Éditeur de code (IDE) . . . . .	4
4 Qu'est ce que Flutter ? . . . . .	5
5 Les avantages de Flutter . . . . .	5
<b>II Installation de Flutter</b>	<b>7</b>
1 Installation Linux . . . . .	7
2 Installation Windows . . . . .	9
2.1 Via téléchargement . . . . .	9
3 Installation Mac . . . . .	9
4 En plus . . . . .	9
<b>III Programmation orientée objet</b>	<b>10</b>
1 Concept de base . . . . .	10
2 Classes et objets . . . . .	11
3 Principe d'encapsulation . . . . .	11
4 Héritage de classe . . . . .	12
5 Classes et méthodes abstraites . . . . .	13
<b>IV Tour d'horizon de Flutter</b>	<b>14</b>
1 Dart . . . . .	14
2 Widgets . . . . .	14
2.1 Comment utiliser un widget . . . . .	14
2.2 Les paramètres children, child et body . . . . .	14
2.3 StatelessWidget . . . . .	14
2.4 StatefulWidget . . . . .	15
2.5 Définir ses propres Widgets . . . . .	15
2.6 Widgets bons à connaître . . . . .	15
3 Packages . . . . .	15
4 Dépendences et pubspec.yaml . . . . .	16
5 Tester votre app . . . . .	16
6 Hot reload . . . . .	16
7 Installer l'app sur votre téléphone . . . . .	16
<b>V Commandes à connaître</b>	<b>17</b>
<b>VI Packages utiles</b>	<b>17</b>



<b>VII</b>	<b>Conseils et bonnes pratiques</b>	<b>18</b>
<b>VIII</b>	<b>A toi de jouer !</b>	<b>19</b>





# I. Introduction

## 1 Prélude


Vous avez toujours rêvé de créer une application mobile innovante, mais les défis techniques vous ont freiné ? Qu'il s'agisse de concevoir une interface élégante, d'intégrer des fonctionnalités fluides ou de coder de manière robuste et maintenable, le développement d'applications mobiles peut sembler être un parcours semé d'embûches. Sans oublier les tests rigoureux nécessaires pour garantir une expérience utilisateur optimale sur une multitude de dispositifs.


C'est là que Flutter  entre en jeu, offrant une solution révolutionnaire qui transforme ces défis en opportunités ! Imaginez un framework<sup>1</sup> qui vous permet de créer des applications esthétiques, performantes et multi-plateformes avec une seule base de code. Flutter  n'est pas seulement un outil, c'est une porte ouverte vers l'innovation dans le monde du développement mobile.

## 2 Prérequis

Il n'y a pas de prérequis stricts, à part un ordinateur évidemment. Toutefois, des connaissances de base en programmation, en particulier dans des langages orientés objet comme Java  ou C++ , sont un atout considérable.

## 3 Éditeur de code (IDE)


Vous pouvez créer des applications avec Flutter  à l'aide de n'importe quel éditeur de texte ou environnement de développement intégré (IDE) combiné aux outils de ligne de commande.

L'utilisation d'un IDE avec une extension ou un plugin Flutter  permet la complétion du code, la coloration syntaxique, l'assistance à l'édition des widgets, le débogage et d'autres fonctionnalités.

Pour cette formation, nous recommandons<sup>2</sup> vivement l'utilisation de **VS Code** car cela simplifie pas mal de choses telle que l'installation de Flutter SDK que nous verrons un peu plus tard.

---

1. Un **framework** est un ensemble cohérent d'outils et de bibliothèques fournissant une structure et des fonctionnalités de base pour le développement d'applications.

2. Utilisez VS Code ou conséquences 

## 4 Qu'est ce que Flutter ?

Flutter est un framework open-source développé par Google pour la création d'interfaces utilisateur (UI). Il permet de concevoir des applications natives multiplateformes pour mobile, web et desktop à partir d'une seule base de code. Google a conçu ce framework afin de simplifier le développement d'applications mobiles, même pour ceux qui ne possèdent pas de vastes connaissances en programmation.



## 5 Les avantages de Flutter

- **Développement multiplateforme** : Avec un seul code, vous pouvez développer une application compatible avec Windows, Linux, macOS, Android, iOS et le web. Vous n'aurez que rarement à vous soucier des problèmes de compatibilité.
- **Approche orientée objet** : Flutter utilise une approche orientée objet, intuitive pour le développement d'applications. Si vous n'êtes pas familier avec cette approche, vous pouvez regarder [cette vidéo](#) pour en avoir une idée. Mais ne vous inquiétez pas, il n'y a pas besoin d'être un crack en OOP pour suivre cette formation et développer de jolies applications !
- **Langage Dart** : Flutter utilise son propre langage de programmation, Dart, conçu pour être facile à apprendre et à utiliser.
- **Système de widgets** : Le système des widgets permet de développer facilement des interfaces, de la plus simple à la plus complexe. Chaque élément de l'interface est un widget que vous pouvez personnaliser et composer.
- **Richesse des packages** : Si vous avez une idée de fonctionnalité, vous pouvez rechercher parmi les nombreux packages disponibles pour voir si un package existant répond déjà à vos besoins, ce qui vous fait gagner du temps.
- **Documentation et support** : Flutter offre une documentation claire et complète, des tutoriels vidéo et même [une chaîne YouTube](#) dédiée. Vous pouvez accéder à la documentation officielle [ici](#).
- **Testing en temps réel** : Vous pouvez tester votre application en temps réel directement sur votre ordinateur, que ce soit en tant que programme, site web ou même sur une simulation de téléphone grâce à Android Studio.
- **Hot reload** : Flutter facilite le débogage avec le hot reload, une fonctionnalité qui permet d'appliquer les changements dans votre code sans devoir fermer et relancer votre application.

- **Conversion en application mobile** : Vous pouvez facilement convertir votre code en une application utilisable sur votre téléphone.



## II. Installation de Flutter

• **NE PAS INSTALLER DART SEPARÉMENT DE FLUTTER. SI VOUS INSTALLEZ FLUTTER, IL SE CHARGE D'INSTALLER DART**

• Cliquez [ici](#) pour le guide officiel d'installation, sur lequel nous nous baserons.

Choisissez la section correspondant à votre système d'exploitation :



Linux : SECTION **1**



Windows : SECTION **2**



MacOS : SECTION **3**

### 1 Installation Linux

Etant donné la diversité des distributions Linux, nous n'allons ici couvrir que la plus commune : **Ubuntu**, plus précisément **Ubuntu 22.04**.

Si vous utilisez une autre distribution, l'installation devrait être identique en majeure partie.

1. Pour commencer, ouvrez votre terminal et vérifiez que ces différents outils sont installés :  
bash, file, mkdir, rm, which grâce à la commande

```
which bash file mkdir rm which
```

2. Mettez à jour la liste des packages :

```
sudo apt-get update -y && sudo apt-get upgrade -y
```

3. Installez les packages suivants : curl, git, unzip, xz-utils, zip, libglu1-mesa

```
sudo apt-get install -y curl git unzip xz-utils zip libglu1-mesa
```

4. Pour les applications Android, nous avons besoin d'Android Studio. Installez donc donc les packages prérequis suivants :

```
sudo apt-get install \
libc6:amd64 libstdc++6:amd64 \
libbz2-1.0:amd64 libncurses5:amd64
```

5. Cliquez [ici](#) pour télécharger la dernière version d'Android Studio.



6. Décompressez le fichier .zip téléchargé vers un emplacement approprié pour vos applications (par exemple, dans /usr/local/ pour votre profil utilisateur ou /opt/ pour les utilisateurs partagés).

7. Pour lancer Android Studio, accédez au répertoire android-studio/bin/, puis exécutez :

```
studio.sh
```

8. Indiquez si vous souhaitez importer les anciens paramètres Android Studio, puis cliquez sur **OK**.

9. Suivez l'**assistant de configuration** d'Android Studio (ce qui implique de télécharger les composants du SDK Android requis pour le développement).

10. Ouvrez un terminal, et entrez :

SURTOUT PAS SNAP

```
sudo snap install flutter --classic
```

11. Tapez **flutter** pour finir l'installation.

Pour passer à la suite





## 2 Installation windows

### 2.1 Via téléchargement

1. **Attention, vous devez avoir Windows 10 ou plus récent !**
2. Choisissez l'installation Windows, puis Android.
3. Téléchargez le fichier "**flutter\_window\_[version].zip**".
4. Déplacez le dans "**C :\\Users\\[votre\_nom\_utilisateur]**".
5. Créez dans "**C :\\Users\\[votre\_nom\_utilisateur]**" un dossier "**dev**", et déplacez-y le fichier .zip. Extrayez les fichiers. Normalement un dossier du même nom a été créé, avec à l'intérieur un dossier "**flutter**".
6. (Redéplacez ce dernier dans "**C :\\Users\\[votre\_nom\_utilisateur]**").
7. Rendez vous dans le dossier "**flutter**" puis "**bin**".
8. Copiez le chemin de fichiers au dessus.
9. Tapez "var" dans la barre de recherche Windows, et cliquez sur le 1er résultat "**Modifier les variables d'environnement du système**".
10. Allez sur "**Paramètres systèmes avancés**", puis "**Variables d'environnement**".
11. Dans la partie "**Variables utilisateur pour [votre\_nom\_utilisateur]**", cliquez sur la variable "**Path**", puis "Modifier".
12. Cliquez sur "Nouveau", et collez le chemin de fichier que vous avez copié plus tôt. Puis cliquez sur "Déplacer vers le haut".
13. Cliquez sur "OK" dans chaque fenêtre. Si vous vous contentez de les fermer avec la croix, les changements ne seront pas pris en compte.
14. Tapez dans la barre de recherche Windows "cmd", et cliquez sur "**Invité de commandes**".
15. Entrez la commande "**flutter -version**". Si vous voyez bien la version de Flutter, vous avez correctement installé Flutter.

## 3 Installation macos

## 4 En plus



## III. Programmation orientée objet

• Avant d'aller plus loin, il faut s'assurer que nous parlons le même langage. Cette section va donc faire un cours **extrêmement** condensé de ce qu'est la programmation orientée objet (Object Oriented Programming, OOP).

Si vous vous y connaissez déjà, où préférez vous concentrer sur Flutter en lui-même vous pouvez passer votre chemin et avancer à la section suivante, mais un rappel ne vous fera pas de mal, si ?

Ce cours express se base en très grosse partie sur le **cours de Bernard Boigelot INFO0062 - Object Oriented Programming (LIEN)**. Ce cours utilise *Java* et non *Dart*, mais *Dart* étant fortement inspiré de *Java*, cela ne devrait pas vous déranger trop.

### 1 Concept de base

• L'**Object Oriented Programming**, que nous allons abrévier en **OOP**, n'est simplement qu'une façon de voir un programme. Ici, un programme est vu comme un ensemble d'**objets** qui interagissent entre eux en se demandant l'un l'autre d'effectuer certaines tâches, une petite équipe qui assure le bon fonctionnement de votre programme.

• Pour faire une analogie, un programme OOP est comme un restaurant : chaque employé a une tâche définie et se contente de faire sa tâche assignée, dans laquelle il est spécialisé. Si vous avez besoin de quelque chose auquel vous n'êtes pas formé, demandez à l'employé qui en est chargé. On aurait jamais l'idée d'envoyer la femme de ménage faire la cuisine ou le barman faire les comptes, non ? Dans votre programme OOP c'est pareil, chaque objet fait sa tâche et fait une requête à un ou plusieurs autres objets si il a besoin de quelque chose qu'il ne sait pas faire.

• De ce fait, en OOP, on met l'accent sur la modularité, c'est à dire d'avoir des fonctionnalités très indépendantes des unes des autres, ce qui permet de :

- Facilement séparer les différentes parties du code qui correspondent aux grandes idées de votre programme ou application.
- Développer une partie d'un programme sans se soucier de savoir ce qui se passe dans d'autres parties.
- Changer une partie d'un programme sans affecter les autres parties
- Réutiliser des morceaux de code ailleurs dans le programme, ou même dans un autre programme
- L'idée générale étant de rendre vos morceaux de codes aussi indépendants que possible tout en leur permettant d'aisément fonctionner ensemble.

• Attention : l'OOP est une vision spécifique de la programmation, indépendante du langage utilisé. Des langages comme *Dart* ou *Java* y sont dédiés, mais vous pouvez appliquer les concepts de l'OOP dans d'autres langages qui n'y sont pas entièrement dédiés, comme *Python*, *C++* ou *PHP*, voir pas du tout comme *C*, *Scala* ou *Golang* (même si évidemment c'est moins recommandé).

## 2 Classes et objets

- Pour l'instant on a appris à séparer des bouts de code dans des fichiers. Rien de très nouveau ou surprenant, donc qu'est-ce qui distingue vraiment l'OOP ?

Ce sont les concepts de **classes** et **d'objets**.

- Un **objet** est une structure de données mêlant à la fois des variables, pour retenir des données, mais aussi des fonctions qu'il peut exécuter, appelées **méthodes**.
- Une **classe** spécifie un objet. Elle définit ses variables et ses méthodes, mais aussi des **constructeurs**, qui permettent de créer cet objet - on dit alors que l'on **instancie** la classe, que l'on crée **une instance** de cette classe. Plus important, la classe permet la séparation entre **l'implémentation**, c'est à dire ce qu'un objet peut faire ou pas, et **l'interface**, comment utiliser cet objet, interagir avec.
- Point important : une classe peut être instanciée en autant d'objet que l'on veut, mais **un objet ne peut avoir qu'une seule classe**.

## 3 Principe d'encapsulation

- La bonne pratique veut que l'on puisse interagir avec un objet qu'à travers son interface. Bien que dans beaucoup de cas pouvoir accéder directement aux variables d'un autre objet soit pratique - et très courant en pratique - le fait de limiter cet accès permet de garantir les aspects de sécurité, par exemple en vérifiant les valeurs passées en arguments.

A cet effet, les variables, méthodes et constructeurs d'une classe peuvent avoir un niveau de visibilité :

- **Public**, n'importe qui peut avoir accès à cette variable/méthode/constructeur.
- **Protégé**, seul les objets de la même classe ou d'une sous classe peuvent avoir accès à cette variable/méthode/constructeur.
- **Privé**, seul cet objet peut avoir accès à cette variable/méthode/constructeur.



## 4 Héritage de classe

- Imaginons que vous êtes engagés pour développer une application permettant la gestion du personnel d'un hôpital. Cet hôpital emploie bien des gens de fonctions diverses. Par exemple vous avez des chirurgiens et des cardiologues. Naturellement, vous créez une classe pour chacun, mais vous rendez compte rapidement qu'elles ont beaucoup de code en commun, voir identique. Il serait bien de pouvoir mettre ce code en un seul point où nos deux classes peuvent y avoir aisément accès.

Si on y pense bien, chirurgien et cardiologue sont tous les deux des médecins.

- C'est pourquoi nous allons définir une troisième classe, la **superclasse** "Médecin" qui regroupe les points communs de "Chirurgien" et "Cardiologue" en seul point. Ensuite, il suffit de spécifier que "Chirurgien" et "Cardiologue" sont des **sous-classes** de "Médecin" pour qu'elles puissent toutes les deux avoir accès au code de médecin, comme si il l'avaient aussi.

On dit que "Médecin" est un **superclasse**, une *généralisation*, alors que "Chirurgien" et "Cardiologue" sont des **sous-classes**, des *spécialisations*. des cas plus spécifiques. "Chirurgien" et "Cardiologue" **héritent** de "Médecin".

- Attention, une *superclasse* peut avoir autant de sous classes que désiré, mais une *sous-classe* **ne peut avoir qu'une seule et unique superclasse**.

- Mais dans notre hôpital d'exemple, il y aussi des employés qui ne sont pas médecins, comme les gardes, les infirmière, les ambulanciers, etc. On peut alors avoir une *super-classe* "Employé" pour englober tout ça. Donc bien qu'un *classe* ne peut avoir qu'une seule *super-classe* directe, elle hérite aussi des **super classes** plus haut dans la hiérarchie. Voyez ça comme un arbre généalogique.

- Remarque : **une sous classe doit pouvoir remplir les mêmes rôles que sa super-classe**. Si j'ai une fonction qui prend en entrée une *objet* de classe "Médecin", je dois pouvoir utiliser cette fonction sans problème avec un *objet* de classe "Chirurgien" ou "Cardiologue".



## 5 Classes et méthodes abstraites

- Changeons encore d'exemple. Vous développez un logiciel de dessin simple, comme Paint, et vous souhaitez pouvoir dessiner des rectangles et des cercles. Vous créez naturellement les classes "Rectangle" et "Circle", et leur attribuez la *méthode* `draw()`, qui prend en entrée une couleur.

Vous vous dîtes que, puisque que les 2 classes ont une méthode `draw()`, il serait sage de créer une *super-classe* "Shape" ou mettre cet méthode. Mais il y a un os : on ne dessine pas un rectangle de la même manière qu'un cercle. On utilise pas les mêmes informations. De ce fait, vous ne pouvez pas déplacer cette *méthode* dans une *super-classe* !

- Le problème ici, est que nous avons la même *interface*, c'est à dire la même utilisation, mais pas la même *implémentation*, c'est à dire le comportement.

Pour remédier à ce problème, nous avons en OOP les **classes abstraites**. Une *classe abstraite* permet de définir l'*interface* d'une méthode **sans** définir son *implémentation*. On parle alors de **méthode abstraite**.

Réglons notre problème : définissons la **classe abstraite** "Shape" avec la **méthode abstraite** `draw()`, prenant en entrée une couleur. Puis, définissons les *sous classes* "Rectangle" et "Circle" avec pour *super-classe* "Shape", ce qui leur permet de définir le code de la *méthode* `draw()` différemment l'un de l'autre.

- Remarques :
  - Les classes abstraites peuvent aussi définir des méthodes normales dont leurs sous classes hériteront.
  - Attention, les sous classes qui héritent d'une classe abstraite **sont obligées d'implémenter ses méthodes abstraites**.
  - Il n'est pas une bonne idée d'avoir une classe abstraite qui hérite d'une classe normale. Pourquoi ?

Une autre méthode pour remédier et le mot-clé **@override**. Quand placé devant une *méthode* que vous héritez, avec la même interface, vous indiquez que vous remplacez l'implémentation de cette *méthode* par celle que vous allez définir ici.



## IV. Tour d'horizon de Flutter

### 1 Dart

- Dart est le langage de programmation que vous utiliserez pour travailler avec Flutter. Sa documentation est extensive, mais nous allons découvrir ensemble quelques détails importants, pour programmeur confirmé, comme débutant.

Notez que la convention d'écriture utilisée pour le code ici et le **CamelCase**, comme souvent en OOP.

### 2 Widgets

- Les Widgets représentent, en large, tout ce que vous pouvez voir/intégrer avec à l'écran. Il s'agit évidemment d'une partie très importante de votre application : le visuel c'est important ! Avoir une interface intuitive et agréable aussi.

Il en existe énormément, que je vous invite à explorer par vous-même. Ici, on se contentera de voir leurs caractéristiques générales, et les exemples plus importants.

#### 2.1 Comment utiliser un widget

- Chaque Widget est associé à une liste de paramètres, qui définiront les caractéristiques de votre widget : sa couleur, sa taille, sa position à l'écran, ou encore son comportement lorsque qu'on interagit avec de telle ou telle manière.

Ces paramètres demandent soit un objet d'une classe spécifique (ou qui hérite de cette classe), soit une fonction, que vous pouvez définir directement, appeler une fonction ou méthode existante.

#### 2.2 Les paramètres children, child et body

- Ces paramètres permettent d'indiquer qu'un Widget contient lui-même un autre Widget. Si un Widget est affiché, tous les widgets qu'il contient le sont également.

**child** et **body** sont rigoureusement identiques, mais **children** lui permet de définir une liste entière de Widgets, (Column et Row sont de bons exemples).

#### 2.3 StatelessWidget

- Les **StatelessWidget** sont tous les Widgets qui ne changeront durant l'utilisation de votre application. Une fois construits, ils ne changent pas, et ce, même si ils utilisent des données variables. Pour le forcer à se reconstruire, il faut soit le détruire et le reconstruire (par exemple en quittant la page), soit *utiliser une autre technique que nous verrons plus tard*.

Ils sont principalement utilisés pour les éléments statiques, comme des titres, des images, etc.

## 2.4 StatefulWidget

- Les **StatefulWidget**, au contraire des *StatelessWidget*, peuvent voir leur **état** changer au cours de l'utilisation de l'app, par exemple parce que vous avez appuyé sur un bouton ou entrez du texte. Ils sont intéressants pour tout ce qui est interactif ou évolue dans le temps, comme la barre de progression d'une vidéo, ou un bouton qui change quand on appuie dessus, des sliders, des menus déroulants

## 2.5 Définir ses propres Widgets

- Il arrivera que aucun Widget existant ne correspondent vraiment à vos besoins, et qu'en plus, vous ayez besoin de le réutiliser à plusieurs endroit. Ce n'est pas grave, vous pouvez créer vous même vos propres Widgets :

- Créez une nouvelle classe (de préférence dans son propre fichier)
- Déclarez là de la façon suivante : **class [nom du widget] extends StatelessWidget/StatefulWidget**

- A partir d'ici, 2 options s'offrent à vous :

- Si vous utilisez un *StatelessWidget*, vous pouvez directement déclarer la méthode suivante : **@override Widget build (BuildContext context)**.
- Si vous utilisez un *StatefulWidget*, les choses sont un peu plus compliquées. Il vous faut une classe *StatefulWidget*, qui sera le widget en lui même, et une classe *State<>*, qui contiendra l'état du Widget et la méthode *build*.

- La méthode *build* que vous avez déclaré est très importante : **elle définit comment construire votre Widget**, à partir de Widget existants, définis par vous-même ou non. La méthode s'attend à ce que vous retourniez un widget, Vous l'aurez compris, la plupart des Widgets sont des assemblages d'autres widgets.

## 2.6 Widgets bons à connaître

- Cette liste n'est pas exhaustive, bien entendu, et sera allongée au fur et à mesure.
  - Scaffold
  - Column et Row

## 3 Packages

- Les packages sont des bibliothèques extérieures, non officielles pour la plupart, qui vous apporteront des fonctionnalités et Widgets supplémentaires. L'avantage est que la plupart d'entre eux sont fort bien documentés. Mais faites tout de même attention à ne pas prendre n'importe lequel ! Regardez le nombre d'installations, et faites des recherches supplémentaires pour voir si il est fréquemment utilisé.

Tout particulièrement, **faites attention avec quelle plateforme le package est compatible !**. Certains marchent sur toutes les plateformes, mais la plupart ne marchent que sur le web, d'autres

que sur Android et iOS, ou que sur MacOS, etc. Si il n'est pas compatible avec la plateforme sur laquelle vous travaillez, pas la peine "d'essayer juste au cas où", au mieux votre code ne compilera pas, et au pire il compile ... avec plus d'un problème caché.

## 4 Dépendances et pubspec.yaml

- Tous les packages sont définis dans le fichier **pubspec.yaml**

## 5 Tester votre app

- La manière la plus simple est, dans un terminal, de se rendre dans le dossier de votre app et d'entrer "**flutter run**".
- Si vous utilisez Visual Studio Code, rendez dans le fichier qui contient la fonction *main()* - généralement *main.dart* - et appuyez sur le petit bouton "play" en haut à droite. Vous aurez alors une barre avec un bouton "stop" pour arrêter l'app, et un bouton "reload" pour redémarrer depuis le début de votre code.
- **flutter run -d web-server** : vous permet de lancer l'app dans un navigateur. La commande vous demandera peut être de choisir quel navigateur vous souhaitez utiliser, après quoi soit l'app s'ouvrira directement sur votre navigateur, soit elle vous donnera une adresse en *localhost* que vous pourrez cliquer. Attention, pour voir les changements dans votre code, il faudra recharger la page.

## 6 Hot reload

- Si vous voulez voir un changement que vous faites dans votre code, pas besoin de fermer l'app : quand vous sauvegarderez votre fichier modifié, les changements seront automatiquement appliqués - sauf si il y a des erreurs syntaxiques dans le fichier. Attention, ça ne vous protège pas des erreurs de sémantique ou de logique : vous devez savoir ce que vous faites !

## 7 Installer l'app sur votre téléphone

- La commande **flutter build apk** vous donnera un fichier .apk, qui est une application utilisable sur Android. Transférez ce fichier sur votre téléphone - peu importe comment - et lancez le. Attention, vous aurez des avertissements de sécurité car votre app ne vient pas de l'appstore, mais vous pouvez les ignorer sans problèmes.





## V. Commandes à connaître

- Voici l'essentiel des commandes Flutter à connaître pour pouvoir travailler tranquillement
  - **flutter create [nom\_de\_projet]** : crée un nouveau projet flutter.
  - **flutter run** : Dans le dossier de votre app, lance l'application.
  - **flutter doctor** : quand quelque chose ne va pas avec votre installation flutter, c'est le premier réflexe à avoir.
  - **flutter clean** : quand le dossier build pose problème, cette commande le nettoie.

## VI. Packages utiles

- Dans cette section, nous vous présentons des packages très utiles pour des fonctionnalités qu'on voudrait souvent avoir sans avoir à se casser la tête à le faire soi même. Cette liste n'est pas (du tout) exhaustive, faites aussi vos propres recherches :

—



## VII. Conseils et bonnes pratiques

---



## VIII. A toi de jouer !

- Maintenant, tu vas devoir recréer le site de N-HiTec en Flutter ! Relax, un squelette te sera fournit ... mais pas plus. N'hésite pas à poser des questions !

