



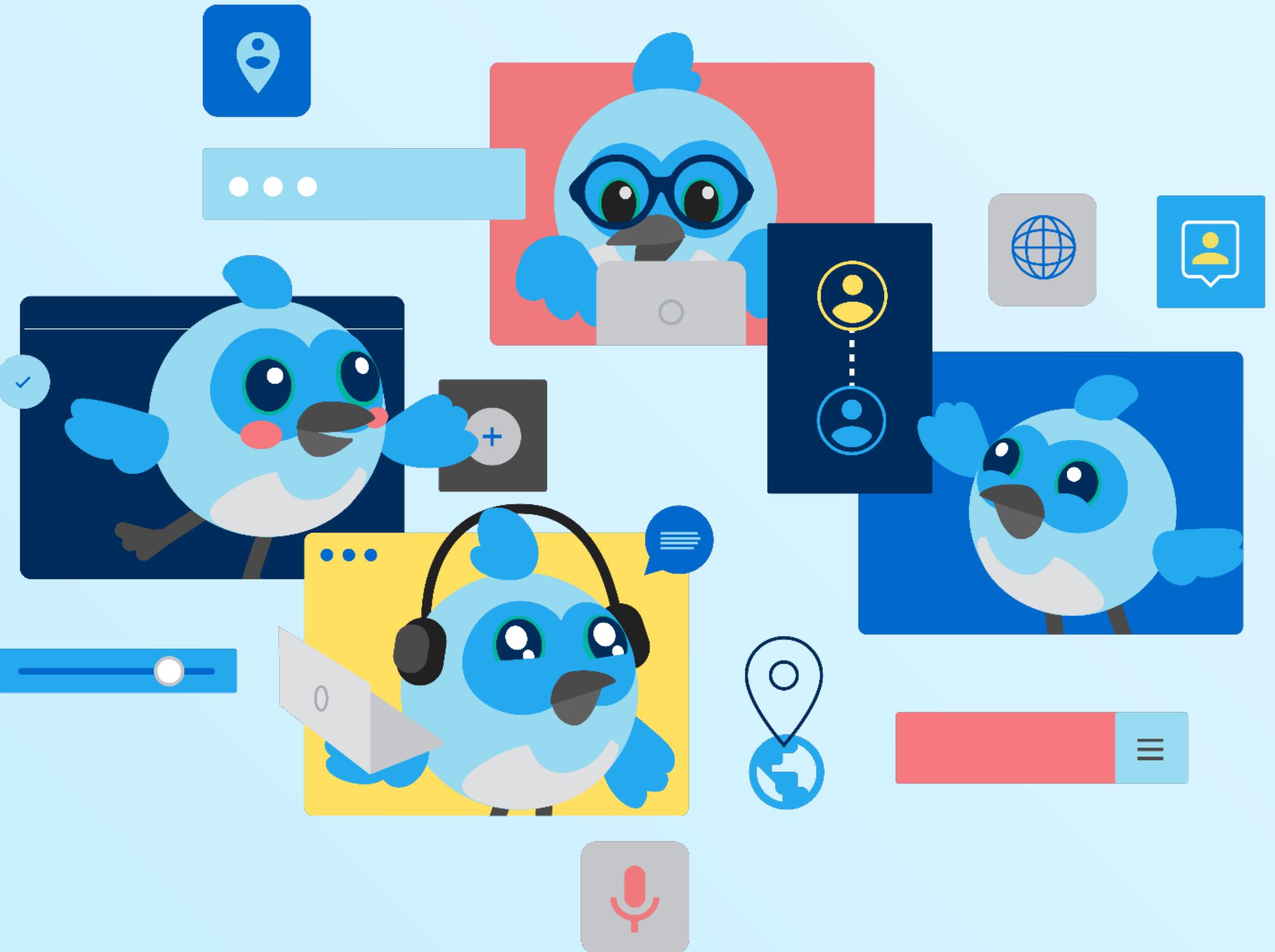
Gestion des exceptions et des erreurs en Dart

Ndongo Tonux SAMB



Agenda

- 1 Dart/Flutter c'est quoi???
- 2 Les Exception en Dart
- 3 Gestion des exceptions avec try/catch
- 4 Crédit d'exceptions personnalisées
- 5 Gestion des erreurs dans Flutter
- 6 Best practices



About me



<http://tonuxcorp.dev>



Flutter

Dakar

```
return Scaffold(  
  body: Person(  
    child: Column(  
      children: [  
        Welcome(),  
        Informations(  
          name: 'Ndongo SAMB',  
          alias: 'Tonux',  
          title: 'IT Consultant',  
        ),  
        From(  
          country: 'Senegal', 🇸🇳  
          capital: 'Dakar'),  
        Location(  
          country: 'Canada', 🇨🇦  
          province: 'Quebec',  
          city: 'Montreal'  
        )  
        Work(  
          label: 'World Anti Doping Agency'  
        )  
      ]  
    );  
);
```

Dart?

- Langage de programmation open-source développé par Google
- Conçu pour le développement d'applications: web , mobile et desktop

1

Orienté Object

Utilise des objets et
des classes.

2

Compilé

Peut être compilé en
code natif et
JavaScript.

3

Asynchrone

Gestion facile des
opérations non
bloquantes.

4

Fortement typé

Système de types
statiques et
dynamiques.

5

Productivité des
développeurs

Syntaxe simple, null
safety, extensions,
mixins.

* Dart est le langage principal de **Flutter**,
permettant de créer des interfaces utilisateur modernes et réactives pour plusieurs plateformes avec une base de code unique.



Dakar

Flutter?

- Framework open-source développé par Google.
- Développement multiplateforme : Android, iOS, web, et bureau.
- Utilise le langage de programmation Dart.

1

Lancé par Google
en mai 2017

2

Compilé

Peut être compilé en
code natif et
JavaScript.

3

Widgets

Éléments de base de
l'interface utilisateur.

4

Flutter Engine

Gère le rendu et les
animations

5

Dart Virtual
Machine

Permet l'exécution du
code Dart.

- Développement multiplateforme avec une seule base de code.
- Performances natives grâce à la compilation en code natif.
- Hot Reload : Modifications instantanées sans redémarrage.
- Bibliothèque de widgets riche pour des interfaces utilisateur attractives.



C'est quoi une exception ou Erreur?

Exception :

- Un événement anormal qui interrompt le flux normal d'un programme.
- Géré par des mécanismes de traitement des exceptions.
- Exemples : Division par zéro, accès à un index hors limites d'un tableau.

Erreur :

- Un problème grave souvent hors du contrôle du programmeur.
- Souvent lié à l'environnement d'exécution.
- Exemples : Manque de mémoire, erreurs de pile (stack overflow).

Pourquoi les Gérer ?

- **Robustesse** : Augmente la stabilité et la fiabilité des applications.
- **Expérience Utilisateur** : Prévenir les plantages et fournir des messages d'erreur clairs.
- **Maintenance** : Facilite le débogage et la maintenance du code.



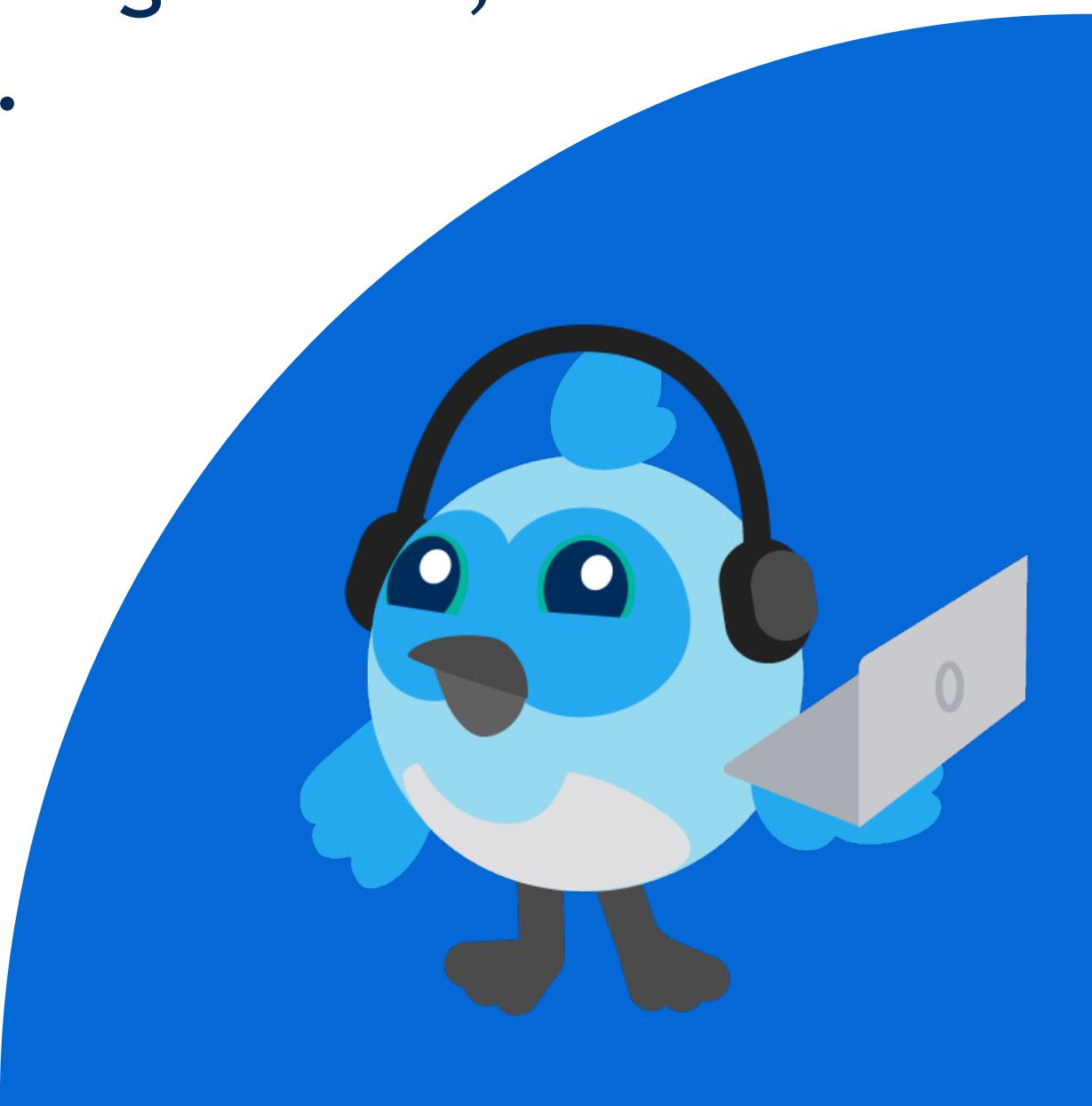
Exceptions et Erreurs en Dart

Les exceptions et les erreurs sont deux types de problèmes qui peuvent survenir lorsque vous exécutez votre code Dart.

Les exceptions sont généralement causées par des opérations non valides, telles que la division par zéro, l'accès à une valeur NULL ou l'appel d'une méthode sur un objet incorrect.

Les erreurs sont plus graves et indiquent une défaillance de la logique du programme, telles que des erreurs de syntaxe, des erreurs de type ou des erreurs d'assertion.

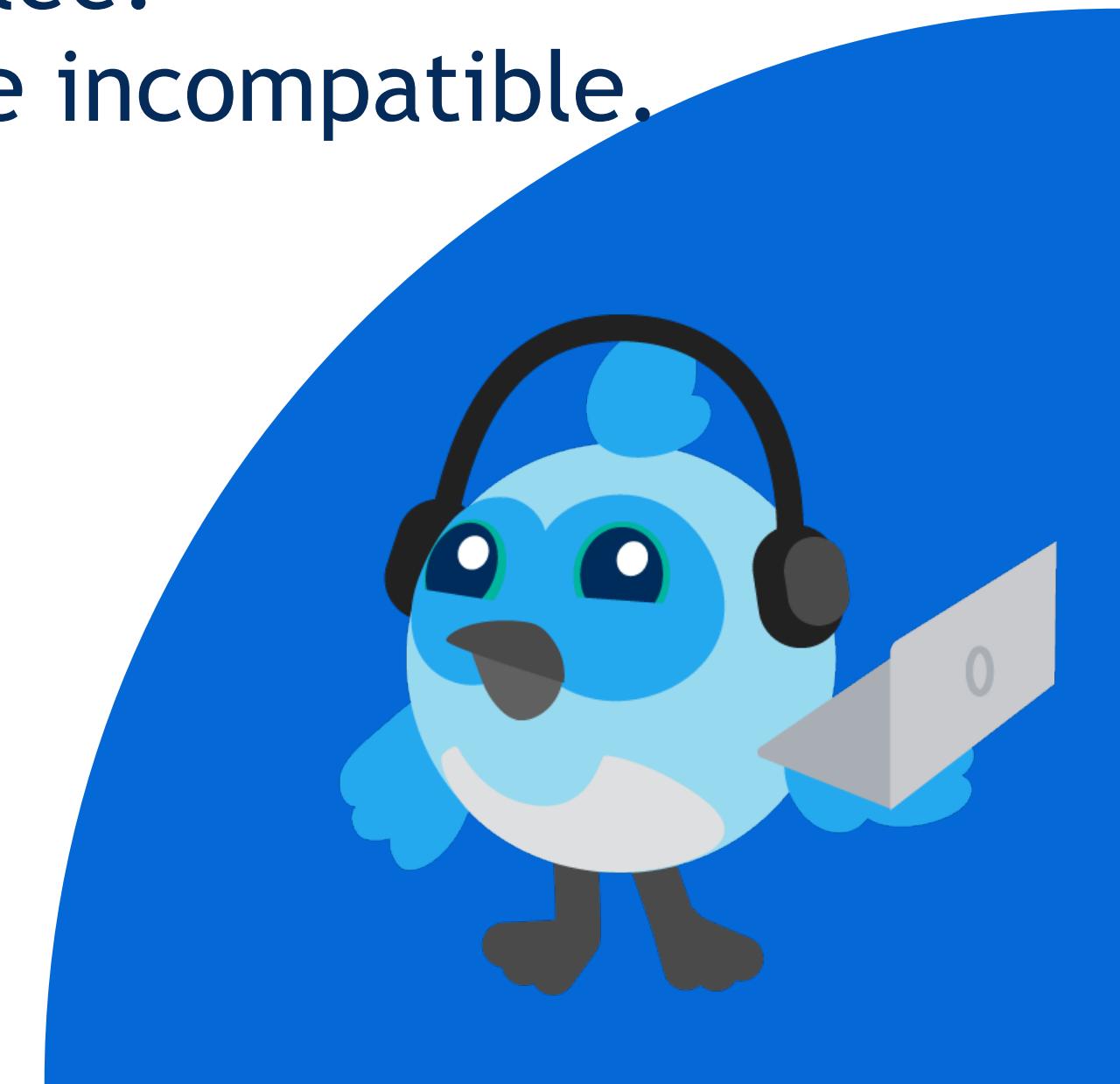
- * Les exceptions et les erreurs peuvent être levées et interceptées à l'aide des mots-clés `throw` et `catch` dans Dart.



Types d'exceptions dans Dart

Dart dispose de plusieurs classes d'exceptions intégrées que vous pouvez utiliser. Certaines des exceptions les plus courantes incluent :

- **FormatException** : levée lorsqu'une opération de conversion de chaîne échoue.
- **ArgumentError** : levé lorsqu'un argument de fonction n'est pas valide.
- **RangeError** : levée lorsqu'une valeur est en dehors de sa plage valide.
- **NoSuchMethodError** : levée lorsqu'une méthode inexistante est appelée.
- **TypeError** : levée lorsqu'une opération rencontre une valeur d'un type incompatible.



Comment lever des exceptions et des erreurs ?

Vous pouvez lever une exception ou une erreur à l'aide du mot-clé `throw`, suivi d'une expression qui renvoie à un objet qui implémente l'interface `Throwable`.

Cette interface est implémentée par les classes intégrées `Exception` et `Error`, ainsi que par leurs sous-classes.

Exemple: vous pouvez lever une exception `FormatException` si vous rencontrez un format de chaîne non valide, ou une `RangeError` si vous accédez à un index hors limites.

Vous pouvez également créer vos propres exceptions et erreurs personnalisées en étendant la classe `Exception` ou `Error` et en remplaçant la méthode `toString`.



Comment détecter les exceptions et les erreurs ? 1/2

Vous pouvez intercepter une exception ou une erreur à l'aide du mot-clé `catch`, suivi d'un bloc de code qui gère le problème.

Vous pouvez éventuellement spécifier le type d'exception ou d'erreur que vous souhaitez intercepter, ainsi qu'un nom de variable pour accéder à l'objet levé. Par exemple, vous pouvez intercepter une exception `FormatException` et afficher son message à l'aide de la syntaxe suivante :

```
try {  
    // some code that might throw a FormatException  
} catch (e, s) { // e is the exception object, s is the stack trace  
    if (e is FormatException) {  
        print(e.message); // print the message of the FormatException  
    }  
}
```



Comment détecter les exceptions et les erreurs ? 2/2

Vous pouvez également utiliser le mot-clé `on` pour spécifier le type d'exception ou d'erreur que vous souhaitez intercepter, sans utiliser de nom de variable. Par exemple, vous pouvez intercepter une erreur `RangeError` et afficher son message à l'aide de la syntaxe suivante :

```
try {  
    // some code that might throw a RangeError  
} on RangeError catch (e) { // e is the exception object  
    print(e.message); // print the message of the RangeError  
}
```



Comment utiliser les blocs finally ?

Vous pouvez utiliser le mot-clé `finally` pour spécifier un bloc de code qui s'exécute toujours après les blocs `try` et `catch`, qu'une exception ou une erreur ait été levée ou non.

Ceci est utile pour nettoyer les ressources, fermer des fichiers ou effectuer toute autre action finale qui doit être effectuée. Par exemple, vous pouvez utiliser un bloc `finally` pour fermer un fichier après l'avoir lu, même si une exception ou une erreur s'est produite pendant le processus de lecture :

```
File file = File('some_file.txt');
try {
    // some code that reads from the file
} catch (e) {
    // some code that handles the exception or error
} finally {
    file.close(); // close the file in any case
}
```



Pourquoi la gestion des erreurs est-elle essentielle ? 1/3

1

Amélioration de
l'expérience utilisateur

2

Empêcher les
applications de se
bloquer

3

Maintenir la stabilité de
l'application

4

Récupération
gracieuse des
défaillances

5

Journalisation et
rapports d'erreurs
(Crashlytics)



Flutter

Dakar

Pourquoi la gestion des erreurs est-elle essentielle ? 2/3

6

Sécurité des nullités et stabilité (Null Safety)

7

Sécurité et conformité

8

Rendre facile les changements

9

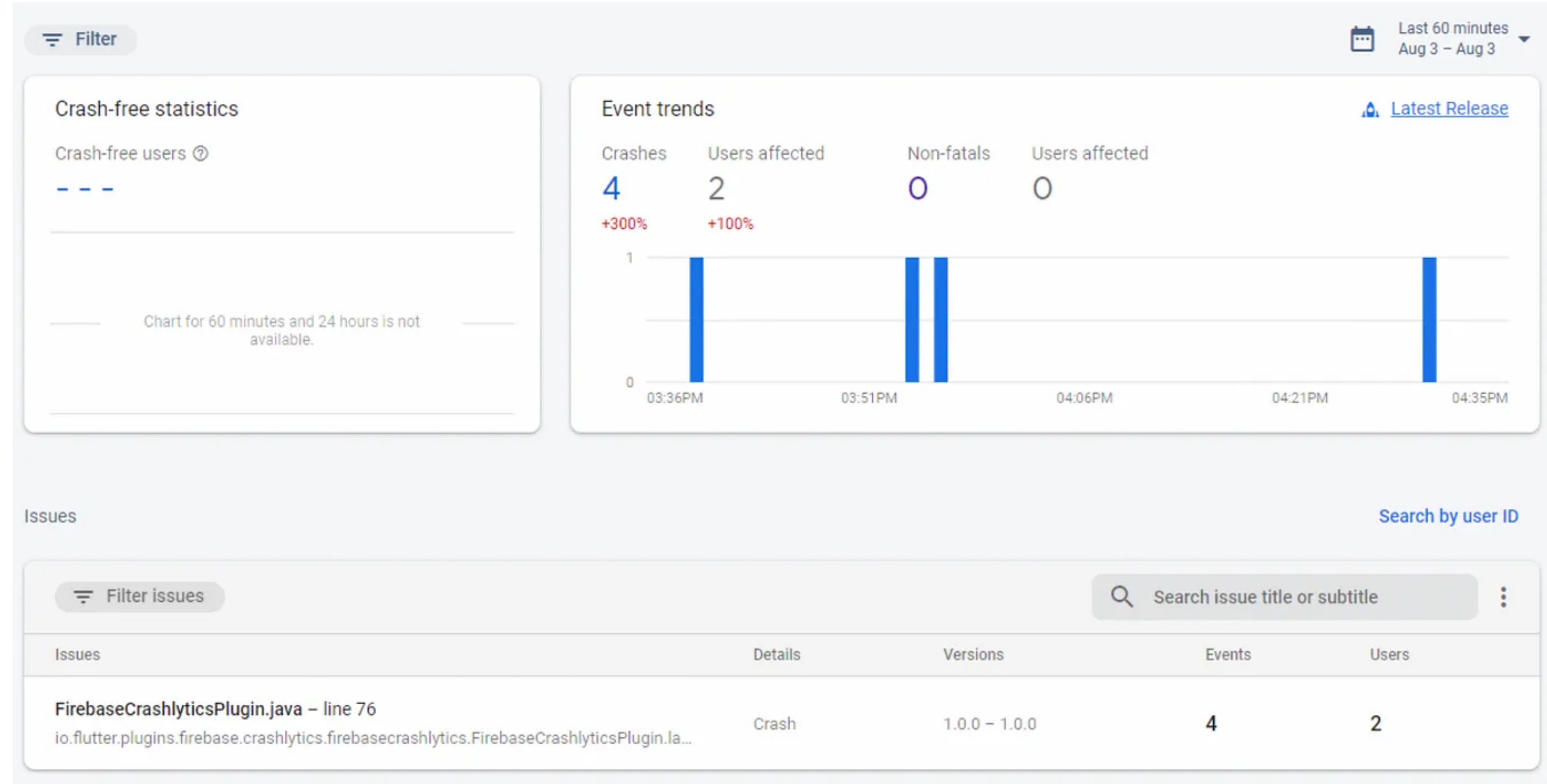
Confiance de l'équipe sur le code

10

Facile pour faire les évolutions



Pourquoi la gestion des erreurs est-elle essentielle ? 3/3



Dakar

Crashlytics

Comment utiliser les blocs finally ?

Vous pouvez utiliser le mot-clé `finally` pour spécifier un bloc de code qui s'exécute toujours après les blocs `try` et `catch`, qu'une exception ou une erreur ait été levée ou non.

Ceci est utile pour nettoyer les ressources, fermer des fichiers ou effectuer toute autre action finale qui doit être effectuée. Par exemple, vous pouvez utiliser un bloc `finally` pour fermer un fichier après l'avoir lu, même si une exception ou une erreur s'est produite pendant le processus de lecture :

```
File file = File('some_file.txt');
try {
    // some code that reads from the file
} catch (e) {
    // some code that handles the exception or error
} finally {
    file.close(); // close the file in any case
}
```

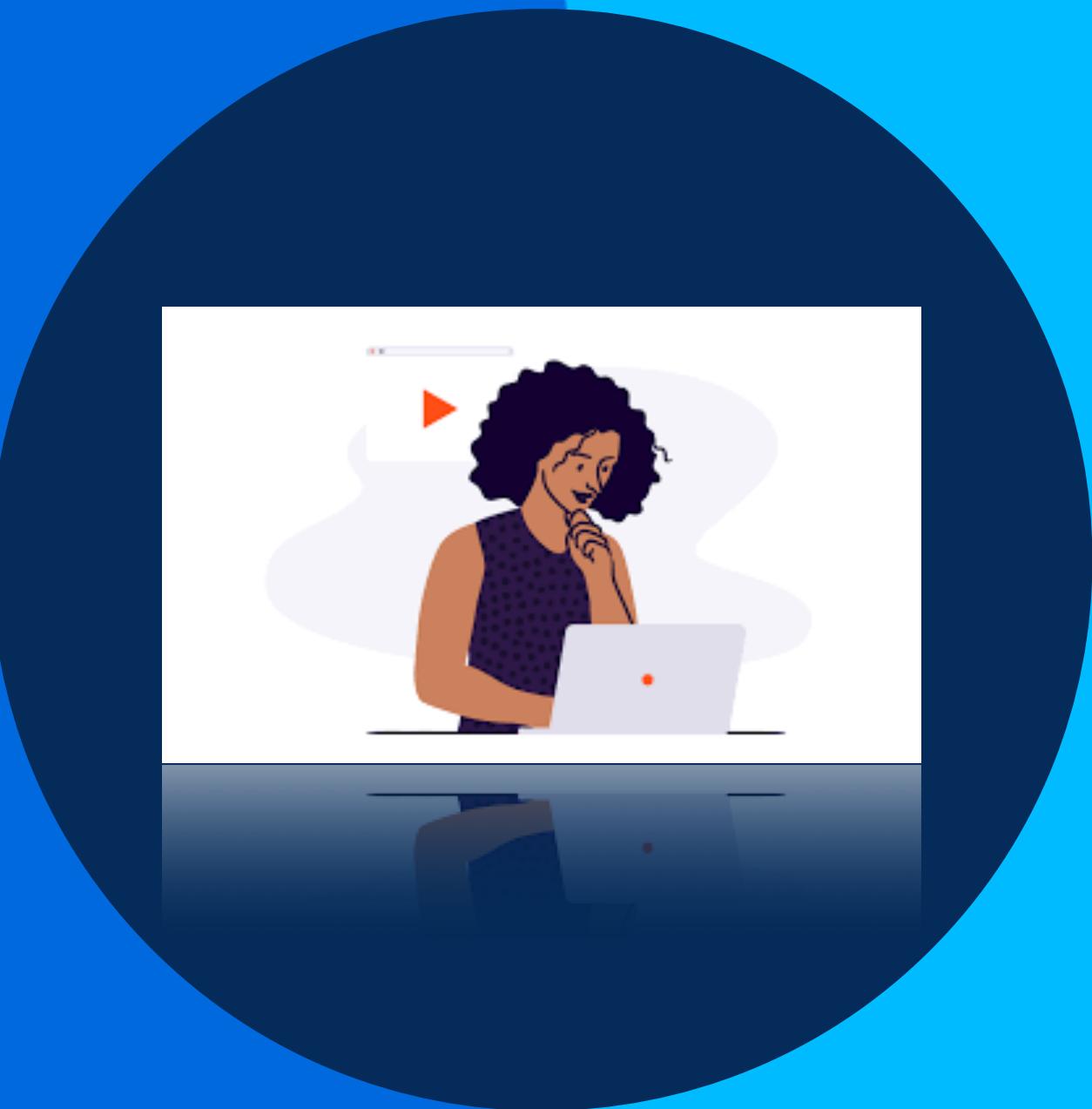


Flutter

Dakar



Demo



THANK YOU!

<http://tonuxcorp.dev>





Accès aux fonctionnalités matérielles du téléphone

Ndongo Tonux SAMB



Agenda

- 1 Introduction
- 2 Vue d'ensemble des fonctionnalités matérielles accessibles
- 3 Permissions et sécurité
- 4 Les plugins Flutter pour les fonctionnalités matérielles
- 5 Cas d'utilisation et meilleures pratiques
- 6 Conclusion et Q&A



About me



<http://tonuxcorp.dev>



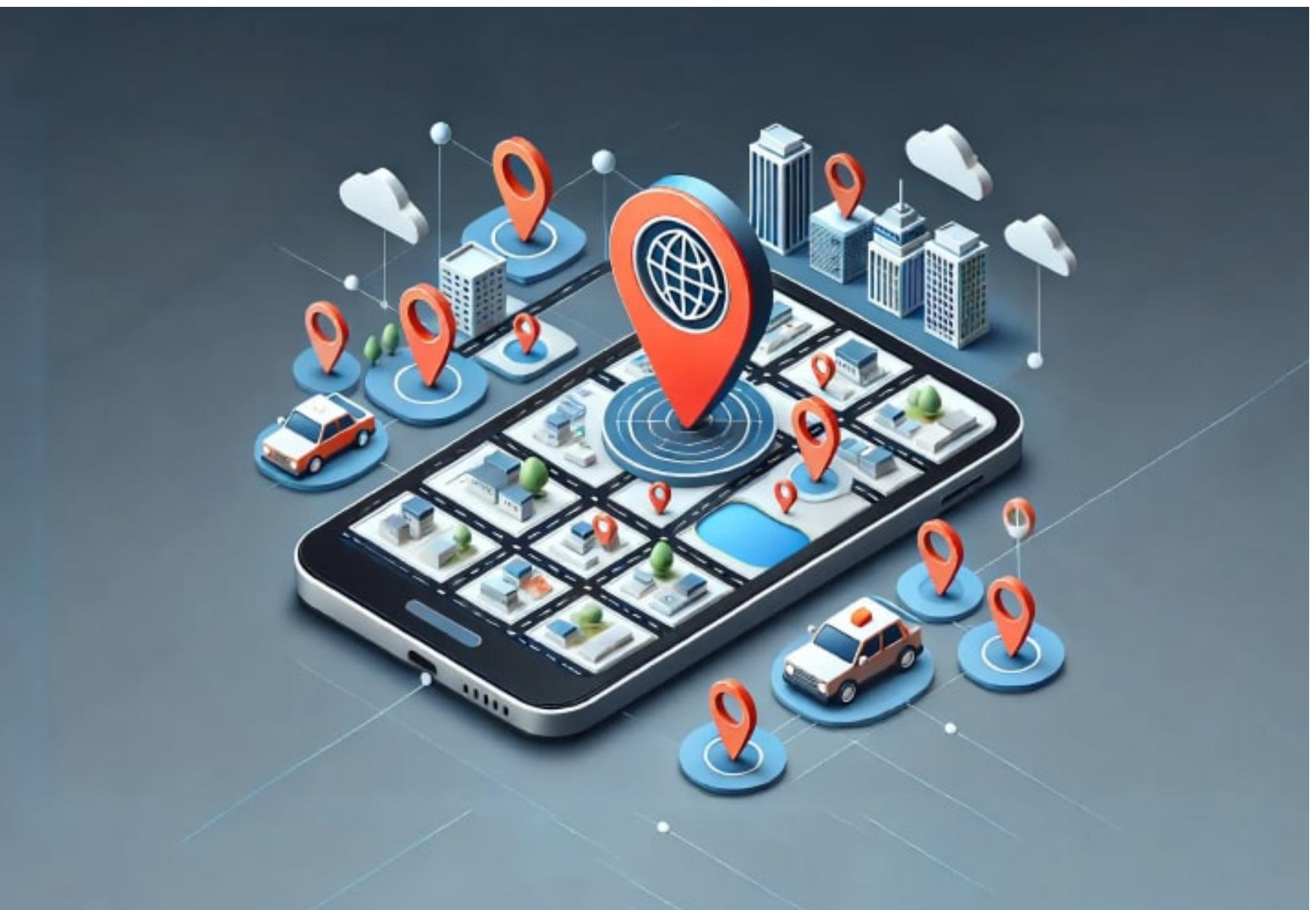
Flutter

Dakar

```
return Scaffold(  
  body: Person(  
    child: Column(  
      children: [  
        Welcome(),  
        Informations(  
          name: 'Ndongo SAMB',  
          alias: 'Tonux',  
          title: 'IT Consultant',  
        ),  
        From(  
          country: 'Senegal', 🇸🇳  
          capital: 'Dakar'),  
        Location(  
          country: 'Canada', 🇨🇦  
          province: 'Quebec',  
          city: 'Montreal'  
        )  
        Work(  
          label: 'World Anti Doping Agency'  
        )  
      ]  
    );  
);
```

C'est quoi une ressource matérielle?

- En développement mobile, une **ressource matérielle** fait référence aux composants physiques d'un appareil mobile (comme un smartphone ou une tablette) qui peuvent être utilisés par une application pour offrir des fonctionnalités avancées. Ces ressources sont essentielles pour permettre aux applications d'interagir directement avec le monde physique.
- Utiliser ces ressources matérielles dans une application nécessite généralement des **permissions** de la part de l'utilisateur, pour garantir la confidentialité et la sécurité de ses données personnelles. Ces autorisations sont souvent demandées la première fois qu'une application tente d'accéder à une ressource matérielle.



Fonctionnalité matérielle?

1

Appareil photo

Permet de prendre des photos, enregistrer des vidéos, ou encore scanner des QR codes ou codes-barres.

2

Microphone

Utilisé pour enregistrer des sons ou pour activer des commandes vocales.

3

Capteurs (accéléromètre, gyroscope)

Mesurent les mouvements et l'orientation de l'appareil, souvent utilisés dans les jeux ou pour détecter des changements de position

4

GPS (géolocalisation)

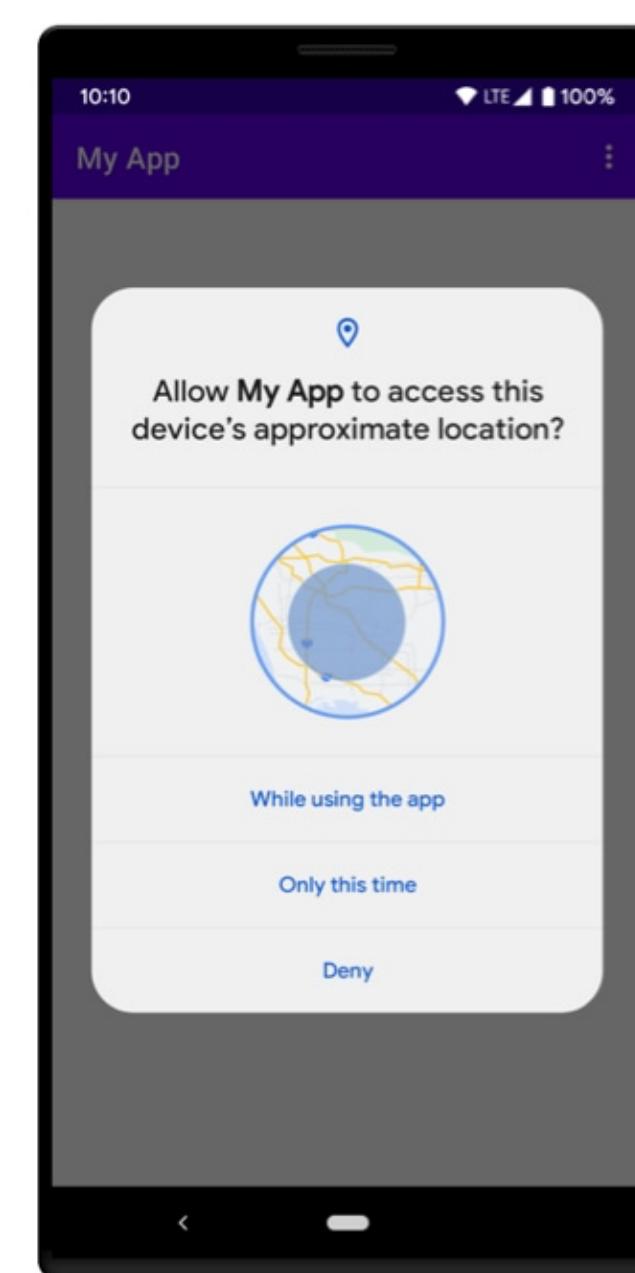
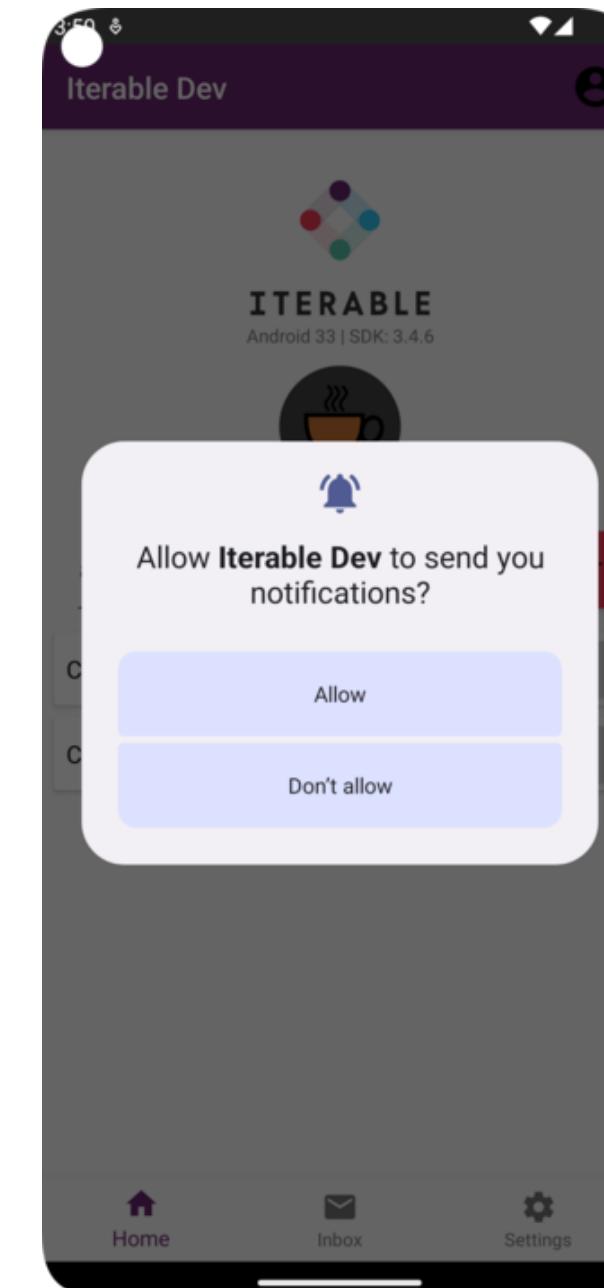
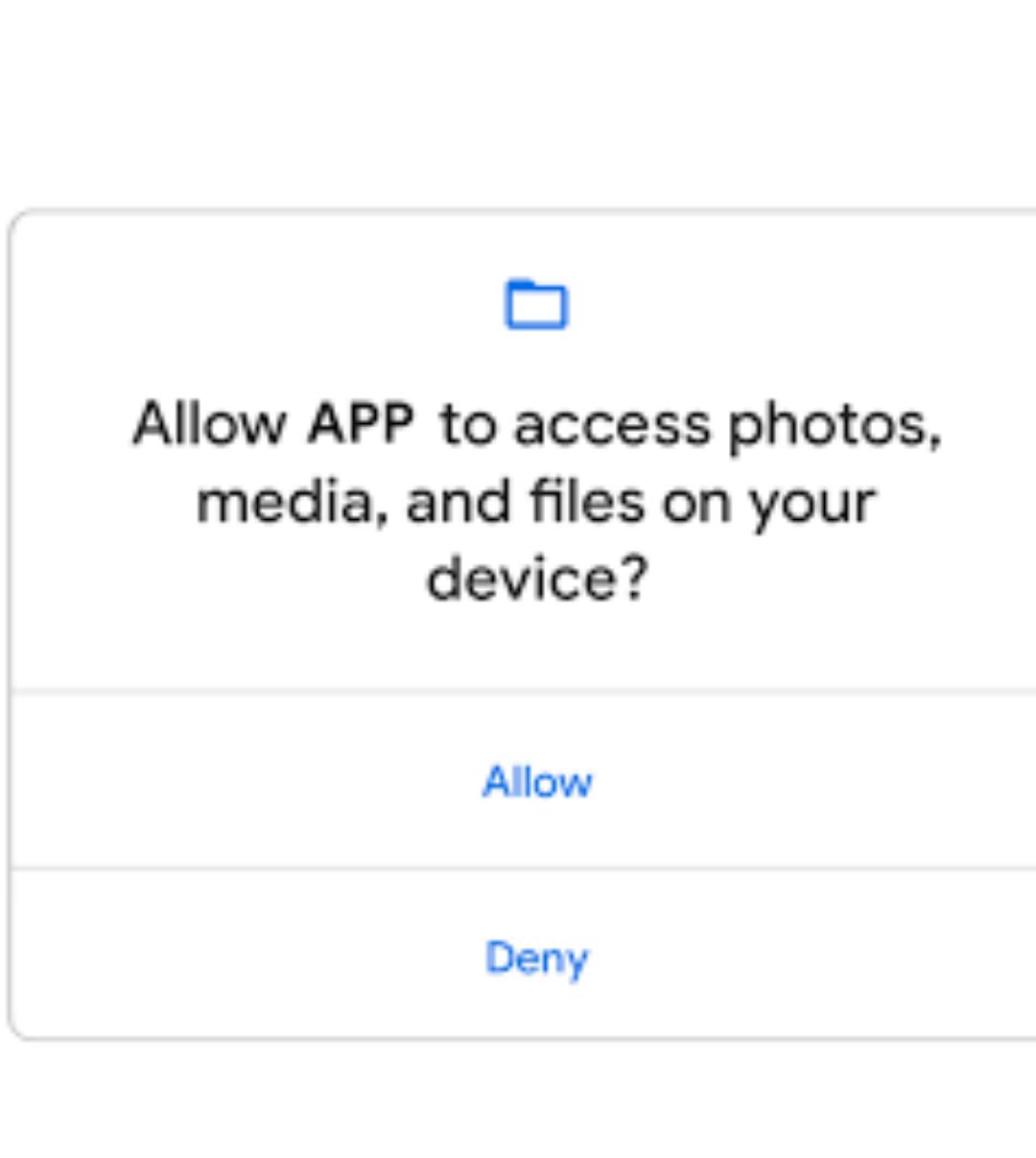
Permet d'obtenir la position géographique de l'utilisateur, utile pour les applications de navigation, de livraison ou de services basés sur la localisation.

5

Bluetooth, NFC, Capteurs biométriques, Capteur de lumière, Capteur de proximité

Permissions

- Importance de demander des permissions (appareil photo, localisation, stockage).
- Configuration dans les fichiers de projet (Android et iOS).
- Bonnes pratiques pour obtenir l'accord des utilisateurs.



Les plugins Flutter pour les fonctionnalités matérielles

- camera : Prendre des photos et enregistrer des vidéos.
- geolocator ou location: Accès à la géolocalisation.
- sensors : Utilisation des capteurs de mouvement.
- permission-handler : Gestion des permissions.



Les plugins Flutter pour les fonctionnalités matérielles

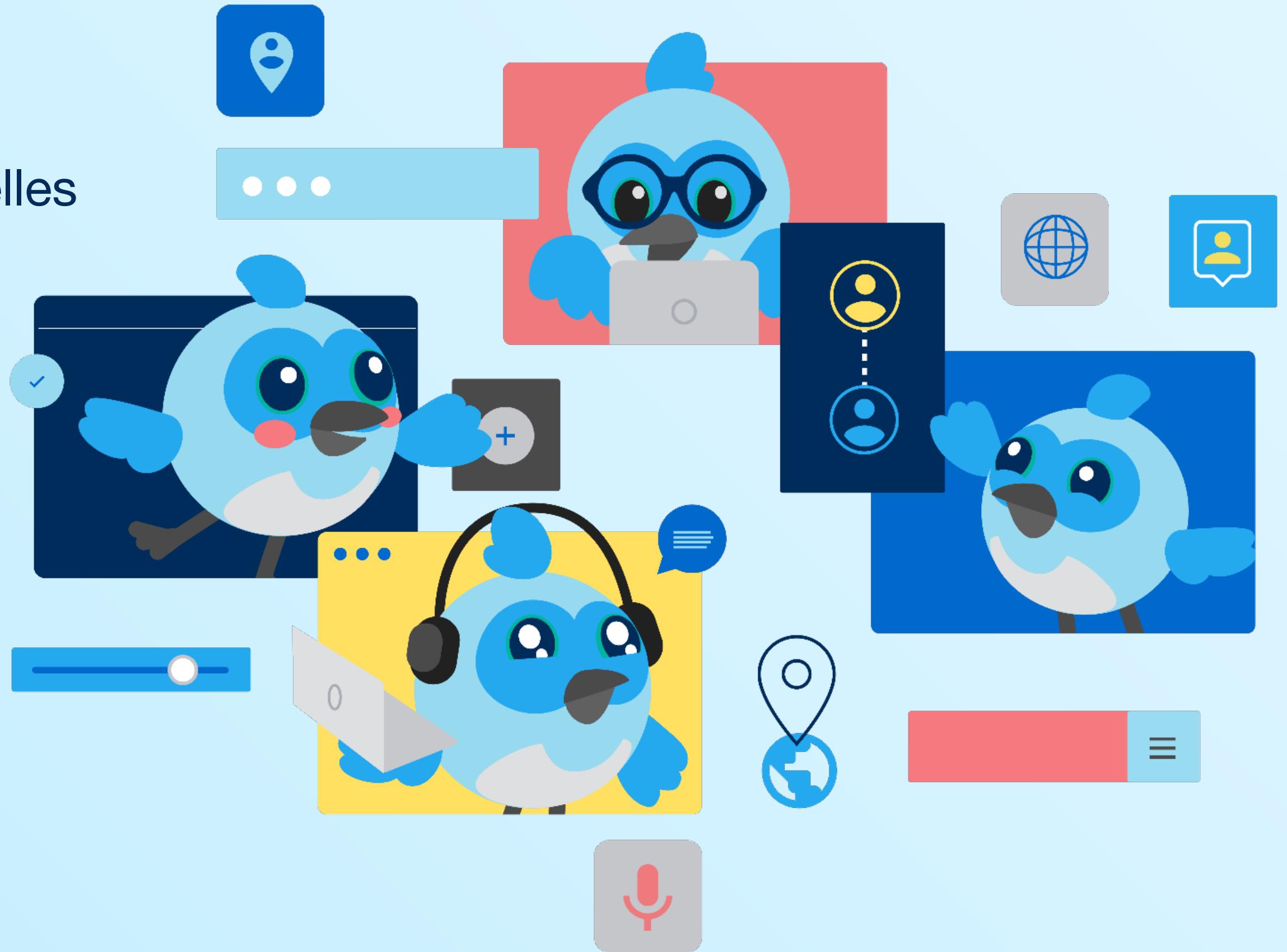
GPS : Géolocalisation en Temps Réel

- **Cas d'utilisation typique :** Applications de navigation (Google Maps, Waze), services de livraison (Uber Eats, DoorDash), ou réseaux sociaux pour des balises de localisation.
- **Meilleures pratiques :**
 - **Utiliser la géolocalisation en arrière-plan avec parcimonie :** N'activer la géolocalisation en arrière-plan que si cela est essentiel à la fonctionnalité, comme pour une application de fitness qui suit un parcours.
 - **Minimiser la fréquence des mises à jour :** Si des mises à jour fréquentes ne sont pas nécessaires, réduisez la fréquence pour économiser la batterie.
 - **Demander la localisation uniquement en cas de besoin :** exemple dans une application de livraison, attendre que l'utilisateur passe une commande pour demander l'autorisation de localisation.
 - **Proposer un message explicatif avant la demande de permission** pour rassurer l'utilisateur quant à la confidentialité de ses données.

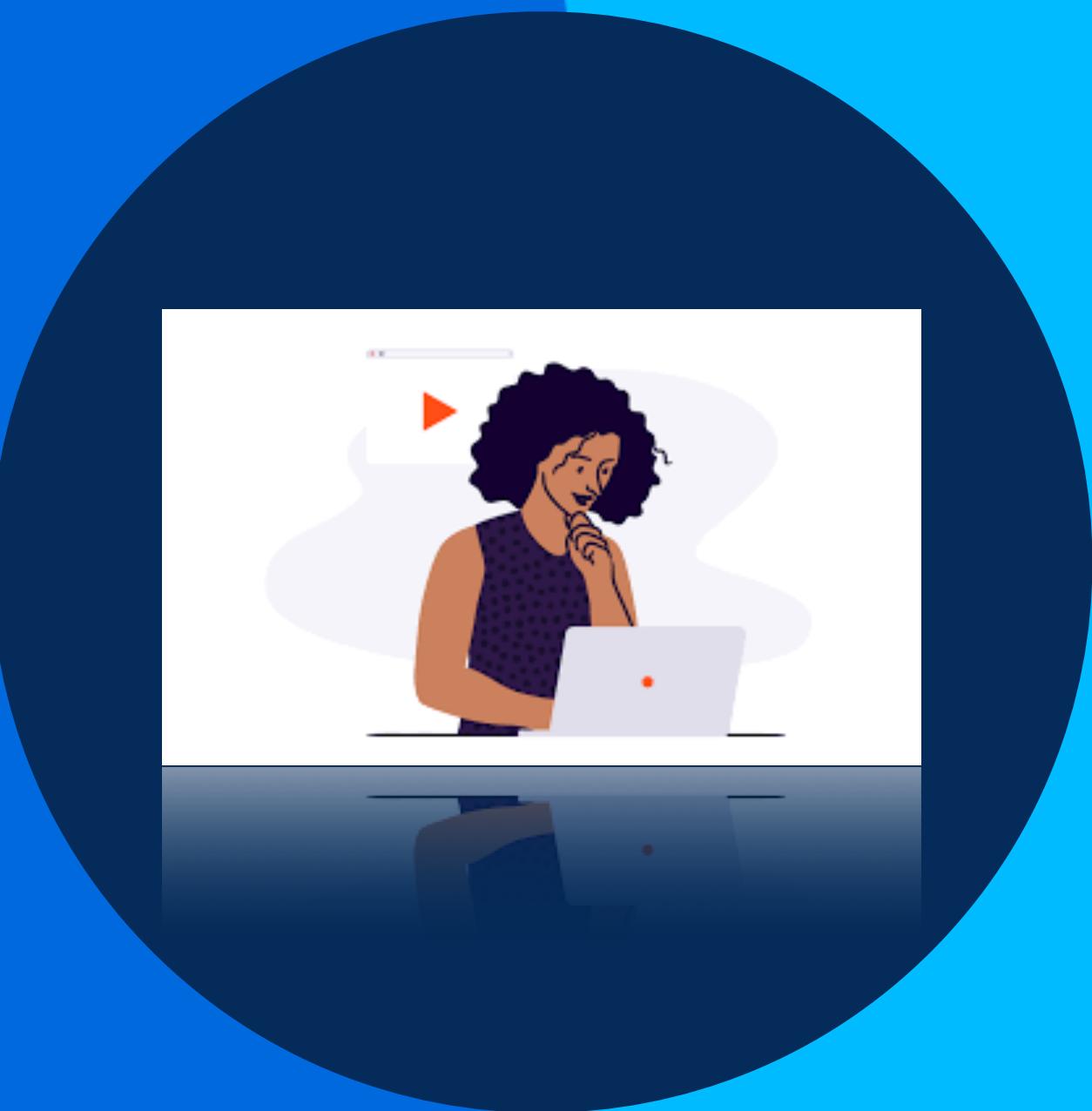


Agenda

- 1 Introduction
- 2 Vue d'ensemble des fonctionnalités matérielles accessibles
- 3 Permissions et sécurité
- 4 Les plugins Flutter pour les fonctionnalités matérielles
- 5 Cas d'utilisation et meilleures pratiques
- 6 Conclusion et Q&A



Demo



THANK YOU!

<http://tonuxcorp.dev>





Animations avancées dans Flutter

Ndongo Tonux SAMB



Agenda

- 1 Introduction
- 2 Implicit Animations
- 3 Explicit Animations
- 4 Animations Using 3rd Party Packages
- 5 Exemples
- 6 Conclusion et Q&A



About me



<http://tonuxcorp.dev>



Flutter

Dakar

```
return Scaffold(  
  body: Person(  
    child: Column(  
      children: [  
        Welcome(),  
        Informations(  
          name: 'Ndongo SAMB',  
          alias: 'Tonux',  
          title: 'IT Consultant',  
        ),  
        From(  
          country: 'Senegal', 🇸🇳  
          capital: 'Dakar'),  
        Location(  
          country: 'Canada', 🇨🇦  
          province: 'Quebec',  
          city: 'Montreal'  
        )  
        Work(  
          label: 'World Anti Doping Agency'  
        )  
      ]  
    )  
  );
```

Animations

Dans une application Flutter, les animations peuvent être classées en deux catégories principales : **animations basées sur le dessin** et **animations basées sur le code**.

- Les **animations basées sur le dessin** concernent des graphiques animés, des vecteurs, des personnages ou tout élément "dessiné", puis animé. Ces animations sont généralement réalisées à l'aide de frameworks ou de packages tiers
- Les **animations basées sur le code**, en revanche, se concentrent sur la mise en page et le style des widgets (listes, couleurs, texte, etc.). Bien que ces animations ne nécessitent pas de packages tiers, elles permettent de créer des animations visuellement impressionnantes, complexes et créatives directement avec Flutter.

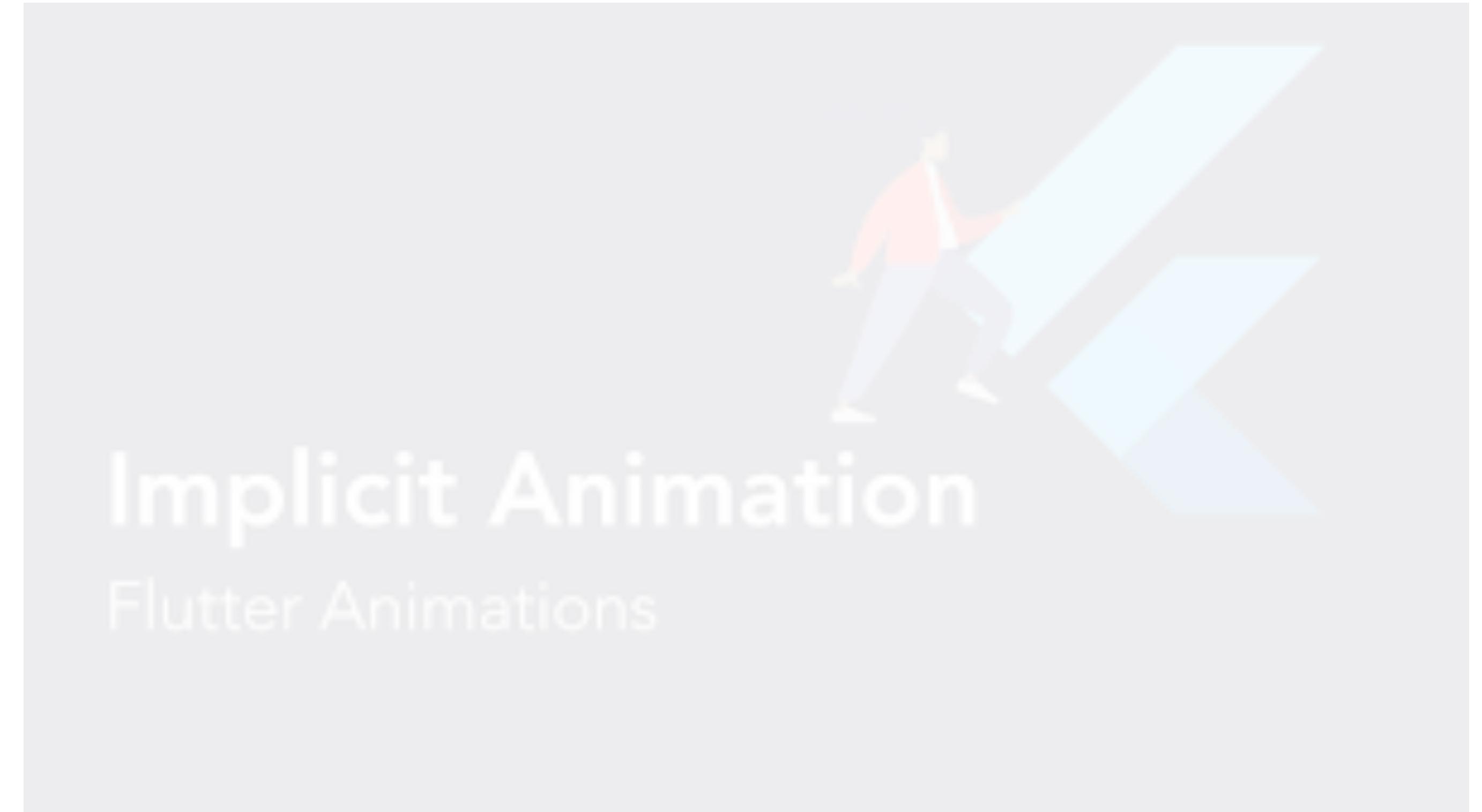


NB: Les animations basées sur le code se divisent en deux types :

- **animations implicites** et
- **animations explicites**.

Implicit Animations

Ce sont les animations les plus simples et les plus faciles à utiliser. Il suffit de changer une valeur pour déclencher une animation et Flutter s'occupe de tout en coulisses.



Flutter

Dakar

Implicit Animations

1.1 Implicit Animations with Ready-to-use widgets

Il s'agit des widgets **AnimatedFoo**, où **Foo** est la propriété animée. La plupart d'entre eux sont des versions animées des widgets que vous connaissez et utilisez déjà, comme **Container/AnimatedContainer**, **Padding/AnimatedPadding**, **Positioned/AnimatedPositioned**, ...etc.



Implicit Animations

1.2. Implicit Animations with TweenAnimationBuilder

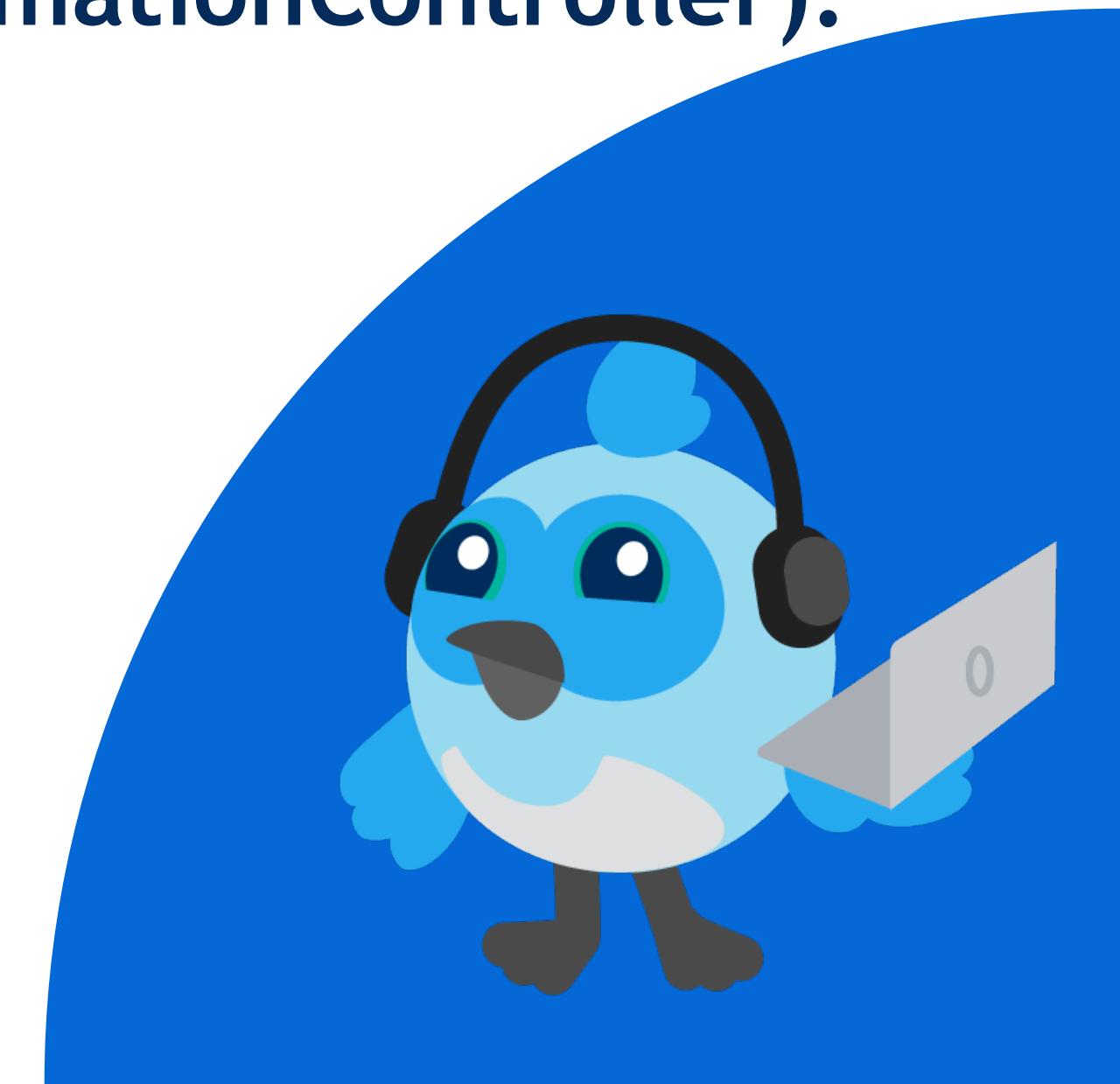
Le **TweenAnimationBuilder** vous permet d'animer implicitement n'importe quelle propriété de n'importe quel widget à l'aide d'une classe **Tween**. La classe Tween tire son nom de « Between » (entre). Elle vous donne essentiellement des valeurs de **début** et de **fin** entre lesquelles animer. Le constructeur du widget **TweenAnimationBuilder** vous donne la valeur animée que vous pouvez appliquer à n'importe quelle propriété des widgets que vous renvoyez dans ce constructeur.



Explicite Animations

Vous souvenez vous que dans les animations implicites, le simple fait de changer une valeur à l'intérieur d'un **AnimatedFoo** ou d'un widget **TweenAnimationBuilder** déclenchaît une animation ?

Les animations explicites ne s'animent pas tant qu'on ne leur demande pas « explicitement » de s'animer. Et vous leur dites de s'animer et comment s'animer, et vous « contrôlez » leur animation à l'aide d'un **contrôleur d'animation (AnimationController)**.



Explicite Animations

2.1. The AnimationController

L'**AnimationController** permet de « contrôler » l'animation. Pour ce faire, il a besoin d'une valeur `vsync` de type **TickerProvider**. Un Ticker garde essentiellement la trace du rendu des images de Flutter et permet au contrôleur de suivre ce ticker et de l'« animer » dans la durée spécifiée, tout en produisant linéairement des valeurs entre les valeurs **lowerBound** et **upperBound** qui sont par défaut 0 & 1.



Explicite Animations

2.2. Explicit Animations with Ready-to-Use Widgets

Ils sont appelés widgets **FooTransition**, où **Foo** est la propriété animée du widget. Certains d'entre eux sont également des widgets animables du widget normal que vous utilisez, par exemple **AlignTransition**, **PositionedTransition**.



Explicite Animations

2.3. Explicit Animations with the AnimatedBuilder widget

[AnimatedBuilder](#) est utile pour les widgets plus complexes qui souhaitent inclure une animation dans le cadre d'une fonction de construction plus large.

Pour utiliser [AnimatedBuilder](#), construisez le widget et passez-lui une fonction de construction.



Explicite Animations

2.2. Explicit Animations with the AnimatedWidget Class

[AnimatedWidget](#) est le plus souvent utilisé avec les objets [Animation](#), qui sont des [objets écoutables](#), mais il peut être utilisé avec n'importe quel [objet écoutable](#), y compris [ChangeNotifier](#) et [ValueNotifier](#).



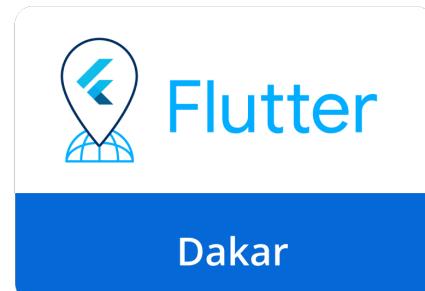
Animations Using 3rd Party Packages

Rive community showcase for ready-to-use cool animations

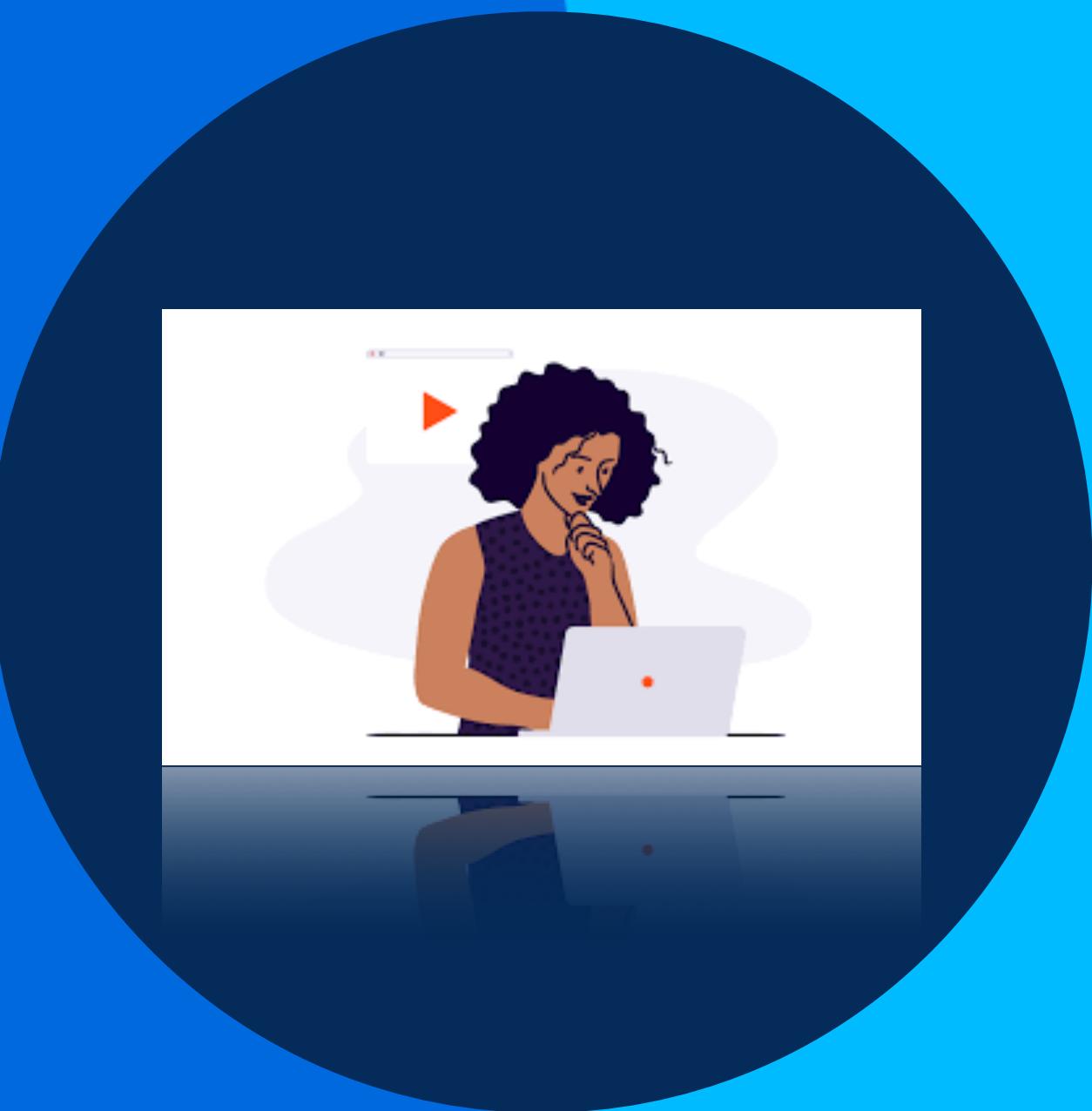
[Rive Flutter Package](#)

[Free LottieFiles animations](#)

[LottieFiles Flutter Package](#)



Demo



THANK YOU!

<http://tonuxcorp.dev>

