Documentation Flutter

Lancez-vous

Configurez vous environnement de développement et commencer à compiler.

Catalogue de Widgets

Plonger dans le catalogue des widgets Flutter disponibles dans le SDK

Documentation API

Visitez l'API de référence de l'environnement Flutter

Quoi de neuf sur flutter.io

5 Novembre 2018

Bienvenue sur le nouveau site Web Flutter!

Nous avons passé les derniers mois à redéfinir le site Web et la manière dont ses informations sont organisées. Nous espérons que vous trouverez plus facilement la documentation que vous recherchez. Parmi les modifications apportées au site Web, citons:

- Mise à jour de la page d'accueil
- Revised page dés références page
- Revised page communauté page
- Nouveau système de navigation à gauche
- Table d'index de page à droite (non affiché sur www.flutterdev.net)

Des nouveaux contenus:

- Une plongée dans les entrailles de Flutter
- Des vidéos techniques
- Une gestion des états
- Les processus Dart en tâche de fond
- Les modes de compilation de Flutter

Si vous avez des questions ou des commentaires sur le nouveau site posez votre question ici..

Quoi de neuf sur Flutter?

Une fois que vous aurez parcouru Lancez-vous, sans oublier Ecrire votre première application Flutter, voilà les étapes suivantes.

Flutter pour les développeurs Android
: Conseils pour les développeurs ayant une expérience du développement Android.

Flutter pour les développeurs iOS
: Conseils pour les développeurs ayant une expérience du développement iOS

Flutter pour les développeurs Web
: Review these HTML -> Flutter analogs if you have web experience.

Flutter pour les développeurs React Native
: Conseils pour les développeurs ayant une expérience du développement React Native

Flutter pour les développeurs Xamarin.Forms
: Conseils pour les développeurs ayant une expérience du développement Xamarin.Forms

Construire des Layouts avec Flutter
: Apprenez à créer des layouts dans Flutter, là où tout n'est que composants

Ajouter de l'interactivité à votre application Flutter
: Apprenez comment ajouter un stateful widget à votre application.

Une visite des widgets du framework Flutter
: En savoir plus sur le framework Flutter’s "react-style".

FAQ
: Get the answers to frequently asked questions.

Vous voulez devenir encore meilleur ?

Quand vous maîtriserez les bases, vous pourrez continuer avec ceci:

Le livres des tutos
: Une collection (en pleine expansion) d'exemples de code couvrant les besoins les plus courants.

Exemples sur GitHub
: Une collection (en pleine expansion) d'applications montrant comment bien utiliser Flutter.

Ajouter des ressources graphiques et des images dans Flutter
    Comment ajouter une ressource dans une application Flutter.
Les animations avec Flutter
    Comment créer et gérer les différents styles d'animation pris en charge par Flutter.
Navigation et routage
    Comment créer gérer la transition (appelée route dans Flutter) vers un nouvel écran*route* in Flutter).
Internationalisation
    Comment rendre votre application internationale.
Utiliser efficacement Dart
    Guides pour écrire une code Dart plus performant.

Sujets Specialisés

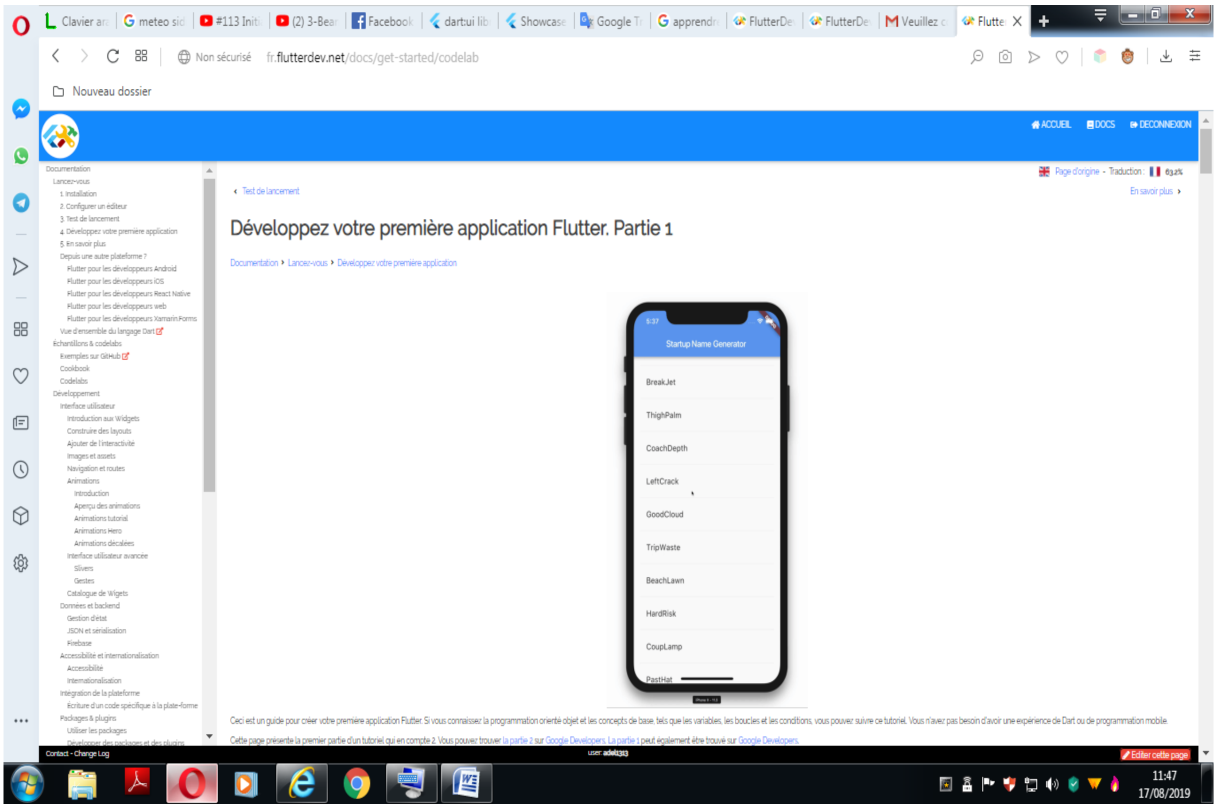
Allez encore plus loin sur les sujets suivants.

L'inspecteur de widgets
    Comment utiliser l'inspecteur de widgets, un outil puissant qui vous permet d'explorer les arborescences de widgets, de désactiver la bannière «DEBUG», d'afficher les informations de performances, et bien d'autres chose.
Polices de caractères personnalisées
    Comment ajouter un nouvelle police de caractères à votre application
Champs de saisie de texte
    Comment configurer et utiliser les champs de saisie de texte de base.
Debugger une application Flutter
    Outils et astuces pour debugger une application.

Ce n'est pas une liste complète des possibilités de Flutter. Utiliser le menu de gauche ou le moteur de recherche pour trouver d'autres sujets.

Développez votre première application Flutter. Partie 1

1. Documentation
2. Lancez-vous
3. Développez votre première application



fr.flutterdev.net/docs/get-started/codelab

- Documentation
  - Lancez-vous
    - 1. Installation
    - 2. Configurer un éditeur
    - 3. Test de lancement
    - 4. Développez votre première application
    - 5. En savoir plus
    - Depuis une autre plateforme ?
      - Flutter pour les développeurs Android
      - Flutter pour les développeurs iOS
      - Flutter pour les développeurs React Native
      - Flutter pour les développeurs web
      - Flutter pour les développeurs Xamarin.Forms
    - Vue d'ensemble du langage Dart
  - Échantillons & codelabs
    - Exemples sur GitHub
    - Cookbook
    - Codelabs
  - Développement
    - Interface utilisateur
      - Introduction aux Widgets
      - Construire des layouts
      - Ajouter de l'interactivité
      - Images et assets
      - Navigation et routes
      - Animations
        - Introduction
        - Aperçu des animations
        - Animations tutorial
        - Animations Hero
        - Animations décalées
      - Interface utilisateur avancée
        - Slivers
        - Gestes
      - Catalogue de Wigets
    - Données et backend
      - Gestion d'état
      - JSON et sérialisation
      - Firebase
    - Accessibilité et internationalisation
      - Accessibilité
      - Internationalisation
    - Intégration de la plateforme
      - Écriture d'un code spécifique à la plate-forme
    - Packages & plugins
      - Utiliser les packages
      - Développer des packages et des plugins

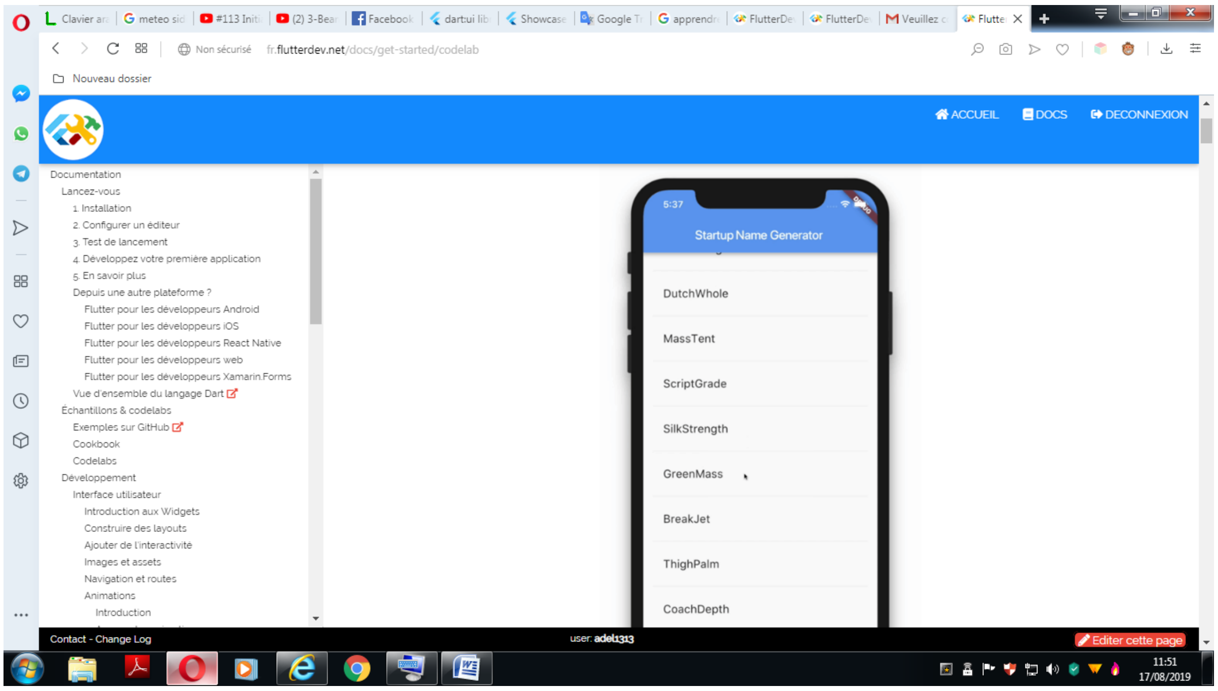
‹ Test de lancement

Développez votre première application Flutter. Partie 1

Documentation › Lancez-vous › Développez votre première application

Ceci est un guide pour créer votre première application Flutter. Si vous connaissez la programmation orienté objet et les concepts de base, tels que les variables, les boucles et les conditions, vous pouvez suivre ce tutoriel. Vous n'avez pas besoin d'avoir une expérience de Dart ou de programmation mobile.

Cette page présente la premier partie d'un tutoriel qui en compte 2. Vous pouvez trouver la partie 2 sur Google Developers. La partie 1 peut également être trouvé sur Google Developers.



Ceci est un guide pour créer votre première application Flutter. Si vous connaissez la programmation orienté objet et les concepts de base, tels que les variables, les boucles et les conditions, vous pouvez suivre ce tutoriel. Vous n'avez pas besoin d'avoir une expérience de Dart ou de programmation mobile.

Cette page présente la premier partie d'un tutoriel qui en compte 2. Vous pouvez trouver la partie 2 sur Google Developers. La partie 1 peut également être trouvé sur Google Developers.

Ce que vous allez développer dans la partie 1

Vous allez implémenter une application mobile simple qui génère et affiche des noms de Start-up. L'utilisateur peut sélectionner et désélectionner des noms et sauver les meilleurs. Le code génère des noms au fur et a mesure que l'utilisateur les fait défiler. Il n'y a pas de limite. L'utilisateur peut faire défiler à l'infini.

Le GIF animé montre le fonctionnement de l'application à la fin de la première partie.

Ce que vous allez apprendre dans la première partie

- Comment écrire une application Flutter qui semble naturelle à la fois sur iOS et Android.
- Structure de base d'une application Flutter.
- Rechercher et utiliser des packages pour étendre les fonctionnalités.
- Utiliser le "rechargement à chaud" pour un cycle de développement plus rapide.

- Comment implémenter un widget avec état (stateful widget).
- Comment créer une liste infinie, chargée progressivement

Dans part 2 de ce codelabs, vous allez ajouter de l'interactivité, modifier le thème de l'application et   ajouter la possibilité de naviguer vers un nouvel écran (ce système de navigation s'appelle *route* dans Flutter

Ce que vous allez utiliser

Vous avez besoin de deux logiciels pour effectuer cet atelier: le SDK Flutter et un éditeur.Ce codelab suppose que vous utilisez Android Studio, mais vous pouvez utiliser votre éditeur préféré.

Vous pouvez faire exécuter le code produit sur l'un des périphériques suivants:

- Un appareil physique (Android ou iOS) connecté à votre ordinateur et mis en mode développeur.
- Un simulateur iOS.
- Un émulateur Android.

Étape 1: Créez l'application Flutter

Créez une application Flutter simple, basée sur un modèle, en suivant les instructions du manuel Getting Started with your first Flutter app.Appelez le projet **startup_namer** (au lieu de *myapp*).

Actuce: Si vous ne voyez pas "Nouveau projet Flutter" comme une option dans votre IDE, assurez-vous que vous avez installé plugins installed for Flutter and Dart.

Dans ce codelab, vous allez principalement éditer le fichier **lib/main.dart** ou se trouve le code principal Dart.

1. Remplacer le contenu de `lib/main.dart`.
   Supprimer tout le code de **lib/main.dart**. Remplacez-le par le code suivant, qui affiche "Hello World" au centre de l'écran.

   lib/main.dart

```
// Copyright 2018 The Flutter team. All rights
reserved.
// Use of this source code is governed by a BSD-style
license that can be
// found in the LICENSE file.

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

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

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

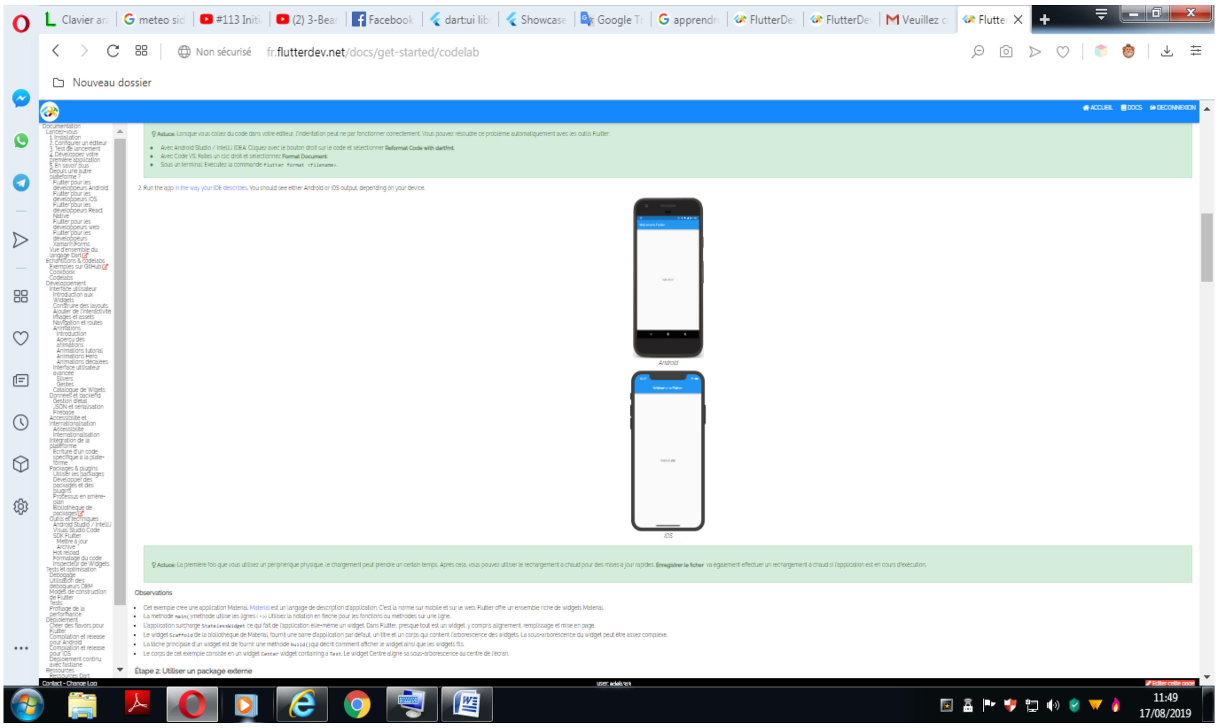
Astuce: Lorsque vous collez du code dans votre éditeur, l'indentation peut ne par fonctionner correctement. Vous pouvez résoudre ce problème automatiquement avec les outils Flutter:

- Avec Android Studio / IntelliJ IDEA: Cliquez avec le bouton droit sur le code et sélectionner **Reformat Code with dartfmt**.
- Avec Code VS: Faites un clic droit et sélectionnez **Format Document**.
- Sous un terminal: Exécutez la commande `flutter format <filename>`.

2. Run the app in the way your IDE describes. You should see either Android or iOS output, depending on your device.

Android

iOS


fr.flutterdev.net/docs/get-started/codelab

Astuce: Lorsque vous collez du code dans votre editeur, l'indentation peut ne par fonctionner correctement. Vous pouvez resoudre ce probleme automatiquement avec les outils Flutter:

- Avec Android Studio / IntelliJ IDEA: Cliquez avec le bouton droit sur le code et sélectionner **Reformat Code with dartfmt**.
- Avec Code VS: Faites un clic droit et sélectionner **Format Document**.
- Sous un terminal: Exécutez la commande `flutter format <filename>`.

2. Run the app in the way your IDE describes. You should see either Android or iOS output, depending on your device.

Android

iOS

Actuce: La première fois que vous utilisez un périphérique physique, le chargement peut prendre un certain temps. Après cela, vous pouvez utiliser le rechargement à chaud pour des mises à jour rapides. **Enregistrer le fichier** va également effectuer un rechargement à chaud si l'application est en cours d'exécution.

Observations

- Cet exemple crée une application Material. Material est un langage de description d'application. C'est la norme sur mobile et sur le web. Flutter offre un ensemble riche de widgets Material.
- La méthode `main()` méthode utilise les lignes (`=>`) Utilisez la notation en flèche pour les fonctions ou méthodes sur une ligne.
- L'application surcharge `StatelessWidget` ce qui fait de l'application elle-même un widget. Dans Flutter, presque tout est un widget, y compris alignement, remplissage et mise en page.
- Le widget `Scaffold` de la bibliothèque de Material, fournit une barre d'application par défaut, un titre et un corps qui contient l'arborescence des widgets. La sous-arborescence du widget peut être assez complexe.
- La tâche principale d'un widget est de fournir une méthode `build()` qui décrit comment afficher le widget ainsi que les widgets fils.
- Le corps de cet exemple consiste en un widget `Center` widget containing a `Text`. Le widget Centre aligne sa sous-arborescence au centre de l'écran.

Étape 2: Utiliser un package externe

Actuce: La première fois que vous utilisez un périphérique physique, le chargement peut prendre un certain temps. Après cela, vous pouvez utiliser le rechargement à chaud pour des mises à jour rapides. **Enregistrer le ficher** va également effectuer un rechargement à chaud si l'application est en cours d'exécution.

Observations

- Cet exemple crée une application Material. Material est un langage de description d'application. C'est la norme sur mobile et sur le web. Flutter offre un ensemble riche de widgets Material.
- La méthode `main()` méthode utilise les lignes (`=>`) Utilisez la notation en flèche pour les fonctions ou méthodes sur une ligne.
- L'application surcharge `StatelessWidget` ce qui fait de l'application elle-même un widget. Dans Flutter, presque tout est un widget, y compris alignement, remplissage et mise en page.
- Le widget `Scaffold` de la bibliothèque de Material, fournit une barre d'application par défaut, un titre et un corps qui contient l'arborescence des widgets. La sous-arborescence du widget peut être assez complexe.
- La tâche principale d'un widget est de fournir une méthode `build()` qui décrit comment afficher le widget ainsi que les widgets fils.
- Le corps de cet exemple consiste en un widget `Center` widget containing a `Text`. Le widget Centre aligne sa sous-arborescence au centre de l'écran.

Étape 2: Utiliser un package externe

Dans cette étape, vous commencerez à utiliser un paquet open-source nommé english_words, qui contient quelques milliers des mots Anglais les plus utilisés ainsi que quelques fonctions pour les utiliser.

Vous pouvez trouver le package `english_words`, ainsi que de nombreux autres logiciels open source, sur le Pub site.

1. Le fichier pubspec gère les actifs et les dépendances d'une application Flutter. Dans `pubspec.yaml`, ajoutez `english_words`(3.1.0 ou supérieur) a la liste de dépendances :

   {step1_base → step2_use_package}/pubspec.yaml

```
  @@ -5,4 +5,5 @@

    dependencies:

    flutter:

    sdk: flutter

    cupertino_icons: ^0.1.2

+ english_words: ^3.1.0
```

2. Lorsque e fichier pubspec est ouvert dans l'éditeur d'Android Studio, Cliquez sur **Packages get**. Cela rapatrie le package dans votre projet. Vous devriez voir ce qui suit dans la console:

```
3. $ flutter pub get
4. Running "flutter pub get" in startup_namer...
5. Process finished with exit code 0
```

   Performing `Packages get` génère aussi automatiquement le fichier `pubspec.lock` avec une liste de tous les paquets rapatriés dans le projet et leurs numéros de version.

6. Dans `lib/main.dart`, importez le nouveau package:

   lib/main.dart

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

Lors de la frappe, Android Studio vous propose des suggestions d'importation de bibliothèques. La chaîne d'importation est ensuite grisée, ce qui vous permet de savoir que la bibliothèque importée est inutilisée (jusqu'à présent).

7. Utilisez le paquet de mots anglais pour générer le texte en utilisant la chaîne "Hello World":

{step1_base → step2_use_package}/lib/main.dart

```
@@ -9,6 +10,7 @@
 class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
+ final wordPair = WordPair.random();
 return MaterialApp(
 title: 'Welcome to Flutter',
 home: Scaffold(
@@ -16,7 +18,7 @@
 title: Text('Welcome to Flutter'),
 ),
    body: Center(
- child: Text(~~'Hello World'~~),
+ child: Text(wordPair.asPascalCase),
    ),
    ),
    );
```

Remarque: La "Casse Pascal" (aussi appelée "upper camel case"), signifie que chaque mot de la chaîne, y compris le premier, commence par une lettre majuscule. Ainsi, “uppercamelcase” devient “UpperCamelCase”.

8. Si l'application est en cours d'exécution, hot reloadpour mettre à jour l'application en cours d'exécution. Chaque fois que vous cliquez sur le rechargement à chaud, ou enregistrez le projet, vous devriez voir une paire de mots différente, choisi au hasard, dans l'application en cours d'exécution. C'est parce que le paire de mots est généré par de la méthode build, qui est exécutée chaque fois que le MaterialApp nécessite un rendu, ou lorsque vous basculez la plate-forme dans l'inspecteur de Flutter.

Android

iOS

Problèmes?

Si votre application ne fonctionne pas correctement, recherchez les fautes de frappe. Si vous voulez essayer certains des outils de débogage de Flutter, Vérifiez DevToolssuite d'outils de débogage et de profilage. Si nécessaire, utilisez le code des liens suivants pour revenir sur la bonne voie.

- pubspec.yaml
- lib/main.dart

Étape 3: Ajouter un widget avec état (stateful widget)

State*less* widgets sont immuables, ce qui signifie que leur les propriétés ne peuvent pas changer - toutes les valeurs sont finales.

State*ful* widgets maintiennent l'état qui pourrait changer pendant la durée de vie du widget. Mise en place d'un widget à état (stateful widget) nécessite au moins deux classes: 1) une classe StatefulWidget qui crée une instance de 2) une classe State. La class StatefulWidget est elle-même immuable, mais la classe State persiste durant toute la vie du widget.

Dans cette étape, vous allez ajouter un widget avec état, `RandomWords`, qui crée sa classe héritée de `State`:`RandomWordsState`. Vous utiliserez alors `RandomWords` en tant qu'enfant dans le widget sans état de `MyApp`.

1. Créez une classe d'état minimale. Ajouter ce qui suit au bas du fichier `main.dart`:

   lib/main.dart (RandomWordsState)

   ```
   class RandomWordsState extends State<RandomWords> {
     // TODO Add build() method
   }
   ```

   Remarquez la déclaration `State<RandomWords>`. Cela indique que nous utilisons le générique State pour l'utiliser avec `RandomWords`. L'important de la logique de l'application et l'état réside ici - il maintient l'état pour le widget `RandomWords`. Cette classe enregistre les paires de mots générés, qui se développent infiniment au fur et à mesure que l'utilisateur fait défiler et les paires de mots préférés (dans la partie 2), au fur et à mesure que l'utilisateur les ajoute ou les supprime de la liste en changeant l'état de l'icône en forme de cœur.

   `RandomWordsState` dépend de la classe `RandomWords`. Vous l'ajouterez ensuite.

2. Ajouter le widget à état `RandomWords` au fichier `main.dart`Le widget `RandomWords` ne fait rien d'autre que créer sa classe State:

   lib/main.dart (RandomWords)

   ```
   class RandomWords extends StatefulWidget {
     @override
     RandomWordsState createState() =>
   RandomWordsState();
   }
   ```

   Après avoir ajouté la classe d'état, l'EDI se plaint que la classe manque une méthode de construction. vous allez devoir ajouter la méthode build qui génère les paires de mots en déplaçant le code de génération de mots depuis `MyApp` vers le classe `RandomWordsState`.

3. Ajouter la méthode `build()` à la classe `RandomWordsState`:

   lib/main.dart (RandomWordsState)

```
class RandomWordsState extends State<RandomWords> {
  @override
  Widget build(BuildContext context) {
    final wordPair = WordPair.random();
    return Text(wordPair.asPascalCase);
  }
}
```

4. Supprimer le code généré dans `MyApp` by making the changes shown in the following diff:

   {step2_use_package → step3_stateful_widget}/lib/main.dart

```
@@ -10,7 +10,6 @@
 class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
- final wordPair = WordPair.random();
 return MaterialApp(
 title: 'Welcome to Flutter',
 home: Scaffold(
@@ -18,8 +17,8 @@
 title: Text('Welcome to Flutter'),
 ),
 body: Center(
- child: ~~Text~~(~~wordPair.asPascalCase~~),
+ child: RandomWords(),
),
),
 );
 }
```

5. Redémarrez l'application. L'application devrait se comporter comme avant, affichant une paire de mots à chaque fois que vous rechargez ou enregistrez l'application à chaud.

Astuce: Si l'avertissement suivant apparaît lors d'un rechargement à chaud, redémarrez l'application:

Reloading…
Some program elements were changed during reload but did not run when the view was reassembled; you might need to restart the app (by pressing "R") for the changes to have an effect.

It might be a false positive, but restarting ensures that your changes are reflected in the app's UI.

Problèmes?

Si votre application ne fonctionne pas correctement, recherchez les fautes de frappe. Si vous voulez essayer certains des outils de débogage de Flutter, Vérifiez DevTools suite of debugging and profiling tools. If needed, use the code at the following link to get back on track.

- lib/main.dart

Étape 4: Créer un ListView à défilement infini

Dans cette étape, vous allez améliorer `RandomWordsState` pour lui permettre de générer et affichez une liste des combinaisons de mots. Au fur et à mesure que l'utilisateur défile, la liste affichée dans un `ListView` se développe à l'infini. `ListView`'s `builder` factory constructor allows you to build a list view lazily, on demand.

1. Add a `_suggestions` list to the `RandomWordsState` pour enregistrer les couples de mots suggérées. Aussi, ajoutez une variable `_biggerFont` qui contiendra le style d'une police de caractère plus grande.

   lib/main.dart

   ```
   class RandomWordsState extends State<RandomWords> {
     final _suggestions = <WordPair>[];
     final _biggerFont = const TextStyle(fontSize:
   18.0);
     // ...
   }
   ```

   Note: Préfixer un identifiant par un trait de soulignement impose la confidentialité in the Dart language.

   Après, vous allez ajouter une fonction `_buildSuggestions()` à la classe `RandomWordsState`. Cette méthode construit le `ListView` qui va proposer les couples de mots.

   La classe `ListView` fournit une propriété `itemBuilder` de la méthode builder. C'est en fait une fonction de construction et de rappel (callback) fournie en tant que fonction anonyme. Deux paramètres sont transmis à la fonction: le `BuildContext` et l'itérateur de ligne, `i`. The iterator begins at 0 and increments each time the function is called. It increments twice for every suggested word pairing: once for the ListTile, and once for the Divider. This model allows the suggested list to grow infinitely as the user scrolls.

2. Ajouter un fonction `_buildSuggestions()` function to the `RandomWordsState` class:

lib/main.dart (_buildSuggestions)

```
Widget _buildSuggestions() {
  return ListView.builder(
      padding: const EdgeInsets.all(16.0),
      itemBuilder: /*1*/ (context, i) {
        if (i.isOdd) return Divider(); /*2*/

        final index = i ~/ 2; /*3*/
        if (index >= _suggestions.length) {

_suggestions.addAll(generateWordPairs().take(10));
/*4*/
        }
        return _buildRow(_suggestions[index]);
      });
}
```

1. La fonction `itemBuilder` callback is called once per suggested word pairing, and places each suggestion into a `ListTile` row. For even rows, the function adds a `ListTile` row for the word pairing. For odd rows, the function adds a `Divider` widget to visually separate the entries. Note that the divider might be difficult to see on smaller devices.
2. Add a one-pixel-high divider widget before each row in the `ListView`.
3. The expression `i ~/ 2` divides `i` by 2 and returns an integer result. For example: 1, 2, 3, 4, 5 becomes 0, 1, 1, 2, 2. This calculates the actual number of word pairings in the `ListView`, minus the divider widgets.
4. If you've reached the end of the available word pairings, then generate 10 more and add them to the suggestions list.

The `_buildSuggestions()` appelle `_buildRow()` un fois par couple de mots. Cette fonction affiche chaque paire dans un `ListTile`, ce qui va vous permettre de rendre les lignes plus attrayantes à l'étape suivante.

3. Add a `_buildRow()` à la classe `RandomWordsState`:

lib/main.dart (_buildRow)

```
Widget _buildRow(WordPair pair) {
  return ListTile(
    title: Text(
      pair.asPascalCase,
      style: _biggerFont,
    ),
  );
}
```

4. In the `RandomWordsState` class, update the `build()` method to use `_buildSuggestions()`, plutôt que d'appeler directement la bibliothèque de mots. (le widget Scaffold implements the basic Material Design visual layout.) Replace the method body with the highlighted code:

   lib/main.dart (build)

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Startup Name Generator'),
    ),
    body: _buildSuggestions(),
  );
}
```

5. In the `MyApp` class, update the `build()` method by changing the title, and changing the home to be a `RandomWords`widget:

   {step3_stateful_widget → step4_infinite_list}/lib/main.dart

   @@ -10,15 +10,8 @@

   10 class MyApp extends StatelessWidget { 10

   11 @override 11

   12 Widget build(BuildContext context) { 12

   13 return MaterialApp(13

   14 - title: '~~Welcome en Flutter~~',

   15 - home: ~~Scaffold~~(

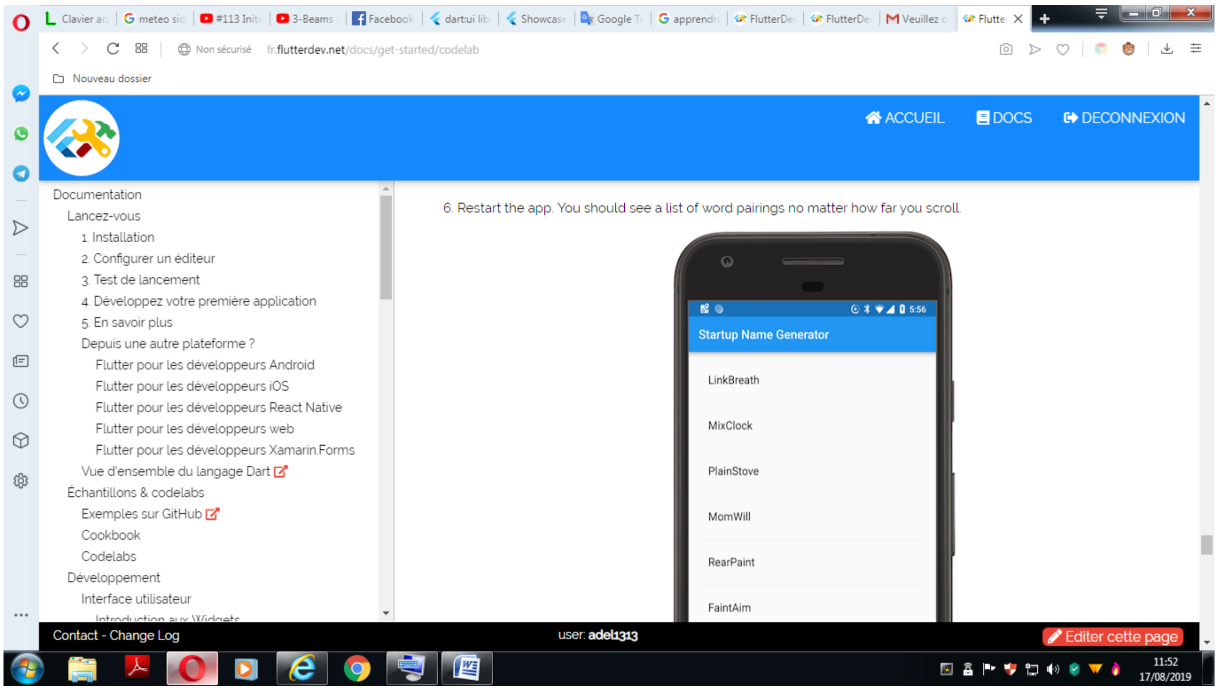
   14 + title: 'Startup Name Generator',

   15 + home: RandomWords(),

```
appBar: AppBar(
title: Text('Welcome to Flutter'),
),
body: Center(
child: RandomWords(),
),
),
);
}
```

6. Restart the app. You should see a list of word pairings no matter how far you scroll.

Android



Non sécurisé fr.flutterdev.net/docs/get-started/codelab

Nouveau dossier

ACCUEIL DOCS DECONNEXION

Documentation
Lancez-vous
1. Installation
2. Configurer un éditeur
3. Test de lancement
4. Développez votre première application
5. En savoir plus
Depuis une autre plateforme ?
Flutter pour les développeurs Android
Flutter pour les développeurs iOS
Flutter pour les développeurs React Native
Flutter pour les développeurs web
Flutter pour les développeurs Xamarin.Forms
Vue d'ensemble du langage Dart
Échantillons & codelabs
Exemples sur GitHub
Cookbook
Codelabs
Développement
Interface utilisateur

6. Restart the app. You should see a list of word pairings no matter how far you scroll.

Startup Name Generator
LinkBreath
MixClock
PlainStove
MomWill
RearPaint
FaintAim

Contact - Change Log
user: adel1313
Editer cette page

iOS

Problèmes?

Si votre application ne fonctionne pas correctement, recherchez les fautes de frappe. Si vous voulez essayer certains des outils de débogage de Flutter, Vérifiez DevTools suite of debugging and profiling tools. If needed, use the code at the following link to get back on track.

- lib/main.dart

Profile or release runs

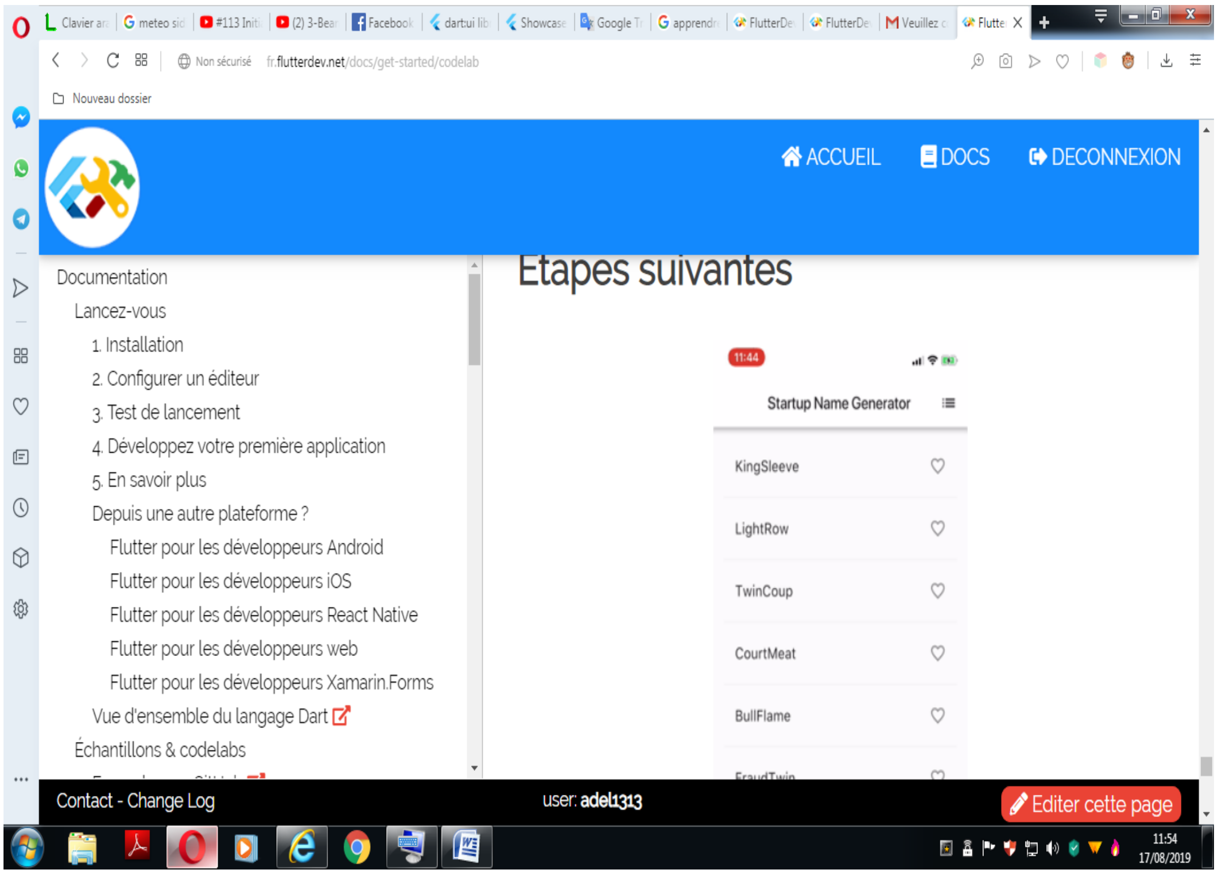
Important: Do *not* test the performance of your app with debug and hot reload enabled.

So far you've been running your app in *debug* mode. Debug mode trades performance for useful developer features such as hot reload and step debugging. It's not unexpected to see slow performance and janky animations in debug mode. Once you are ready to analyze performance or release your app, you'll want to use Flutter's "profile" or "release" build modes. For more details, see Flutter's build modes.

Etapes suivantes

The app from part 2

Toutes nos félicitations!


Non sécurisé fr.flutterdev.net/docs/get-started/codelab

Nouveau dossier

ACCUEIL DOCS DECONNEXION

Documentation
Lancez-vous
1. Installation
2. Configurer un éditeur
3. Test de lancement
4. Développez votre première application
5. En savoir plus
Depuis une autre plateforme ?
Flutter pour les développeurs Android
Flutter pour les développeurs iOS
Flutter pour les développeurs React Native
Flutter pour les développeurs web
Flutter pour les développeurs Xamarin.Forms
Vue d'ensemble du langage Dart
Échantillons & codelabs

Etapes suivantes

11:44
Startup Name Generator
KingSleeve
LightRow
TwinCoup
CourtMeat
BullFlame
FraudTwin

Contact - Change Log
user: adel1313
Editer cette page

11:54
17/08/2019

Vous avez écrit une application interactive Flutter fonctionnant à la fois sur iOS et Android. Dans ce code, vous avez:

- Créé une application Flutter à partir de zéro.
- Ecrit du code Dart.
- Tirer parti d'une bibliothèque externe.
- Utilisé le rechargement à chaud pour un cycle de développement plus rapide.
- Implemented a stateful widget.
- Créé une liste à défilement infinie, chargée progressivement.

Si vous souhaitez étendre cette application, passez à la partie 2 sur le site Google Developers Codelabsoù vous ajouterez les fonctionnalités suivantes:

- Implémentez l'interactivité en ajoutant une icône de cœur cliquable pour enregistrer couples de mots préférés.
- Implémenter la navigation vers un nouvel écran contenant les favoris sauvegardés.
- Modifiez la couleur du thème pour créer une application entièrement blanche.

- Test de lancement

Flutter pour les développeurs Android

This document is meant for Android developers looking to apply their existing Android knowledge to build mobile apps with Flutter. If you understand the fundamentals of the Android framework then you can use this document as a jump start to Flutter development.

Vos connaissances et vos compétences sur Android sont très précieuses lors de la construction avec Flutter, parce que Flutter s'appuie sur le système d'exploitation mobile pour de nombreuses capacités et configurations. Flutter est un nouveau moyen de créer des interfaces utilisateur pour les mobiles, mais il a un système de plug-in pour communiquer avec Android (et iOS) sans interface utilisateur les tâches. Si vous êtes un expert avec Android, vous n'avez pas à tout réapprendre. utiliser Flutter.

This document can be used as a cookbook by jumping around and finding questions that are most relevant to your needs.

Views

What is the equivalent of a View in Flutter?

How is react-style, or *declarative*, programming different than the traditional imperative style? For a comparison, see Introduction to declarative UI.

In Android, the `View` is the foundation of everything that shows up on the screen. Buttons, toolbars, and inputs, everything is a View. In Flutter, the rough equivalent to a `View` is a `Widget`. Widgets don't map exactly to Android views, but while you're getting acquainted with how Flutter works you can think of them as "the way you declare and construct UI".

However, these have a few differences to a `View`. To start, widgets have a different lifespan: they are immutable and only exist until they need to be changed. Whenever widgets or their state change, Flutter's framework creates a new tree of widget instances. In comparison, an Android view is drawn once and does not redraw until `invalidate` is called.

Flutter's widgets are lightweight, in part due to their immutability. Because they aren't views themselves, and aren't directly drawing anything, but rather are a description of the UI and its semantics that get "inflated" into actual view objects under the hood.

Flutter includes the Material Components library. These are widgets that implement the Material Design guidelines. Material Design is a flexible design system optimized for all platforms, including iOS.

But Flutter is flexible and expressive enough to implement any design language. For example, on iOS, you can use the Cupertino widgets to produce an interface that looks like Apple's iOS design language.

How do I update widgets?

In Android, you update your views by directly mutating them. However, in Flutter, `Widget`s are immutable and are not updated directly, instead you have to work with the widget's state.

This is where the concept of Stateful and Stateless widgets comes from. A `StatelessWidget` is just what it sounds like—a widget with no state information.

`StatelessWidgets` are useful when the part of the user interface you are describing does not depend on anything other than the configuration information in the object.

For example, in Android, this is similar to placing an `ImageView` with your logo. The logo is not going to change during runtime, so use a `StatelessWidget` in Flutter.

If you want to dynamically change the UI based on data received after making an HTTP call or user interaction then you have to work with `StatefulWidget` and tell the Flutter framework that the widget's `State` has been updated so it can update that widget.

The important thing to note here is at the core both stateless and stateful widgets behave the same. They rebuild every frame, the difference is the `StatefulWidget` has a `State` object that stores state data across frames and restores it.

If you are in doubt, then always remember this rule: if a widget changes (because of user interactions, for example) it's stateful. However, if a widget reacts to change, the containing parent widget can still be stateless if it doesn't itself react to change.

The following example shows how to use a `StatelessWidget`. A common `StatelessWidget` is the `Text` widget. If you look at the implementation of the `Text` widget you'll find that it subclasses `StatelessWidget`.

```
Text(
  'I like Flutter!',
  style: TextStyle(fontWeight: FontWeight.bold),
);
```

As you can see, the `Text` Widget has no state information associated with it, it renders what is passed in its constructors and nothing more.

But, what if you want to make "I Like Flutter" change dynamically, for example when clicking a `FloatingActionButton`?

To achieve this, wrap the `Text` widget in a `StatefulWidget` and update it when the user clicks the button.

For example:

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}
```

```
class _SampleAppPageState extends State<SampleAppPage> {
  // Default placeholder text
  String textToShow = "I Like Flutter";

  void _updateText() {
    setState(() {
      // update the text
      textToShow = "Flutter is Awesome!";
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(child: Text(textToShow)),
      floatingActionButton: FloatingActionButton(
        onPressed: _updateText,
        tooltip: 'Update Text',
        child: Icon(Icons.update),
      ),
    );
  }
}
```

How do I lay out my widgets? Where is my XML layout file?

In Android, you write layouts in XML, but in Flutter you write your layouts with a widget tree.

The following example shows how to display a simple widget with padding:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: MaterialButton(
        onPressed: () {},
        child: Text('Hello'),
        padding: EdgeInsets.only(left: 10.0, right: 10.0),
      ),
    ),
  );
}
```

You can view the layouts that Flutter has to offer in the widget catalog.

How do I add or remove a component from my layout?

In Android, you call `addChild()` or `removeChild()` on a parent to dynamically add or remove child views. In Flutter, because widgets are immutable there is no direct equivalent to `addChild()`. Instead, you can pass a function to the parent that returns a widget, and control that child's creation with a boolean flag.

For example, here is how you can toggle between two widgets when you click on a `FloatingActionButton`:

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  // Default value for toggle
  bool toggle = true;
  void _toggle() {
    setState(() {
      toggle = !toggle;
    });
  }

  _getToggleChild() {
    if (toggle) {
      return Text('Toggle One');
    } else {
      return MaterialButton(onPressed: () {}, child:
Text('Toggle Two'));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: _getToggleChild(),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _toggle,
        tooltip: 'Update Text',
        child: Icon(Icons.update),
      ),
    );
  }
}
```

How do I animate a widget?

In Android, you either create animations using XML, or call the `animate()` method on a view. In Flutter, animate widgets using the animation library by wrapping widgets inside an animated widget.

In Flutter, use an `AnimationController` which is an `Animation<double>` that can pause, seek, stop and reverse the animation. It requires a `Ticker` that signals when vsync happens, and produces a linear interpolation between 0 and 1 on each frame while it's running. You then create one or more `Animation`s and attach them to the controller.

For example, you might use `CurvedAnimation` to implement an animation along an interpolated curve. In this sense, the controller is the "master" source of the animation progress and the `CurvedAnimation` computes the curve that replaces the controller's default linear motion. Like widgets, animations in Flutter work with composition.

When building the widget tree you assign the `Animation` to an animated property of a widget, such as the opacity of a `FadeTransition`, and tell the controller to start the animation.

The following example shows how to write a `FadeTransition` that fades the widget into a logo when you press the `FloatingActionButton`:

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

void main() {
  runApp(FadeAppTest());
}

class FadeAppTest extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Fade Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyFadeTest(title: 'Fade Demo'),
    );
  }
}

class MyFadeTest extends StatefulWidget {
  MyFadeTest({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyFadeTest createState() => _MyFadeTest();
}
```

```
class _MyFadeTest extends State<MyFadeTest> with
TickerProviderStateMixin {
  AnimationController controller;
  CurvedAnimation curve;

  @override
  void initState() {
    super.initState();
    controller = AnimationController(duration: const
Duration(milliseconds: 2000), vsync: this);
    curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
          child: Container(
              child: FadeTransition(
                  opacity: curve,
                  child: FlutterLogo(
                    size: 100.0,
                  )))),
      floatingActionButton: FloatingActionButton(
        tooltip: 'Fade',
        child: Icon(Icons.brush),
        onPressed: () {
          controller.forward();
        },
      ),
    );
  }
}
```

For more information, see Animation & Motion widgets, the Animations tutorial, and the Animations overview.

How do I use a Canvas to draw/paint?

In Android, you would use the `Canvas` and `Drawable`s to draw images and shapes to the screen. Flutter has a similar `Canvas` API as well, since it is based on the same low-level rendering engine, Skia. As a result, painting to a canvas in Flutter is a very familiar task for Android developers.

Flutter has two classes that help you draw to the canvas: `CustomPaint` and `CustomPainter`, the latter of which implements your algorithm to draw to the canvas.

To learn how to implement a signature painter in Flutter, see Collin's answer on StackOverflow.

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

void main() => runApp(MaterialApp(home: DemoApp()));

class DemoApp extends StatelessWidget {
  Widget build(BuildContext context) => Scaffold(body:
Signature());
}

class Signature extends StatefulWidget {
  SignatureState createState() => SignatureState();
}

class SignatureState extends State<Signature> {
  List<Offset> _points = <Offset>[];
  Widget build(BuildContext context) {
    return GestureDetector(
      onPanUpdate: (DragUpdateDetails details) {
        setState(() {
          RenderBox referenceBox =
context.findRenderObject();
          Offset localPosition =

referenceBox.globalToLocal(details.globalPosition);
          _points =
List.from(_points)..add(localPosition);
        });
      },
      onPanEnd: (DragEndDetails details) =>
_points.add(null),
      child: CustomPaint(painter:
SignaturePainter(_points), size: Size.infinite),
    );
  }
}

class SignaturePainter extends CustomPainter {
  SignaturePainter(this.points);
  final List<Offset> points;
  void paint(Canvas canvas, Size size) {
    var paint = Paint()
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5.0;
    for (int i = 0; i < points.length - 1; i++) {
      if (points[i] != null && points[i + 1] != null)
        canvas.drawLine(points[i], points[i + 1], paint);
    }
  }
  bool shouldRepaint(SignaturePainter other) =>
other.points != points;
}
```

How do I build custom widgets?

In Android, you typically subclass `View`, or use a pre-existing view, to override and implement methods that achieve the desired behavior.

In Flutter, build a custom widget by composing smaller widgets (instead of extending them). It is somewhat similar to implementing a custom `ViewGroup` in Android, where all the building blocks are already existing, but you provide a different behavior—for example, custom layout logic.

For example, how do you build a `CustomButton` that takes a label in the constructor? Create a CustomButton that composes a `RaisedButton` with a label, rather than by extending `RaisedButton`:

```
class CustomButton extends StatelessWidget {
  final String label;

  CustomButton(this.label);

  @override
  Widget build(BuildContext context) {
    return RaisedButton(onPressed: () {}, child:
Text(label));
  }
}
```

Then use `CustomButton`, just as you'd use any other Flutter widget:

```
@override
Widget build(BuildContext context) {
  return Center(
    child: CustomButton("Hello"),
  );
}
```

Intents

What is the equivalent of an Intent in Flutter?

In Android, there are two main use cases for `Intent`s: navigating between Activities, and communicating with components. Flutter, on the other hand, does not have the concept of intents, although you can still start intents through native integrations (using a plugin).

Flutter doesn’t really have a direct equivalent to activities and fragments; rather, in Flutter you navigate between screens, using a `Navigator` and `Route`s, all within the same `Activity`.

A `Route` is an abstraction for a “screen” or “page” of an app, and a `Navigator` is a widget that manages routes. A route roughly maps to an `Activity`, but it does not carry the same meaning. A navigator can push and pop routes to move from screen to screen. Navigators work like a stack on which you can `push()` new routes you want to navigate to, and from which you can `pop()` routes when you want to “go back”.

In Android, you declare your activities inside the app’s `AndroidManifest.xml`.

In Flutter, you have a couple options to navigate between pages:

- Specify a `Map` of route names. (MaterialApp)
- Directly navigate to a route. (WidgetApp)

The following example builds a Map.

```
void main() {
  runApp(MaterialApp(
    home: MyAppHome(), // becomes the route named '/'
    routes: <String, WidgetBuilder> {
      '/a': (BuildContext context) => MyPage(title: 'page
A'),
      '/b': (BuildContext context) => MyPage(title: 'page
B'),
      '/c': (BuildContext context) => MyPage(title: 'page
C'),
    },
  ));
}
```

Navigate to a route by `push`ing its name to the `Navigator`.

```
Navigator.of(context).pushNamed('/b');
```

The other popular use-case for `Intent`s is to call external components such as a Camera or File picker. For this, you would need to create a native platform integration (or use an existing plugin).

To learn how to build a native platform integration, see Developing packages and plugins.

How do I handle incoming intents from external applications in Flutter?

Flutter can handle incoming intents from Android by directly talking to the Android layer and requesting the data that was shared.

The following example registers a text share intent filter on the native activity that runs our Flutter code, so other apps can share text with our Flutter app.

The basic flow implies that we first handle the shared text data on the Android native side (in our `Activity`), and then wait until Flutter requests for the data to provide it using a `MethodChannel`.

First, register the intent filter for all intents in `AndroidManifest.xml`:

```
<activity
  android:name=".MainActivity"
  android:launchMode="singleTop"
  android:theme="@style/LaunchTheme"

android:configChanges="orientation|keyboardHidden|keyboard
|screenSize|locale|layoutDirection"
  android:hardwareAccelerated="true"
  android:windowSoftInputMode="adjustResize">
  <!-- ... -->
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category
android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
</activity>
```

Then in `MainActivity`, handle the intent, extract the text that was shared from the intent, and hold onto it. When Flutter is ready to process, it requests the data using a platform channel, and it's sent across from the native side:

```
package com.example.shared;

import android.content.Intent;
import android.os.Bundle;

import java.nio.ByteBuffer;

import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.ActivityLifecycleListener;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import
io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {

  private String sharedText;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null)
{
      if ("text/plain".equals(type)) {
        handleSendText(intent); // Handle text being sent
      }
    }

    new MethodChannel(getFlutterView(),
"app.channel.shared.data").setMethodCallHandler(
      new MethodCallHandler() {
        @Override
        public void onMethodCall(MethodCall call,
MethodChannel.Result result) {
          if (call.method.contentEquals("getSharedText"))
{
            result.success(sharedText);
            sharedText = null;
          }
        }
      });
  }

  void handleSendText(Intent intent) {
    sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
  }
}
```

Finally, request the data from the Flutter side when the widget is rendered:

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample Shared App Handler',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}
```

```
class _SampleAppPageState extends State<SampleAppPage> {
  static const platform = const
MethodChannel('app.channel.shared.data');
  String dataShared = "No data";

  @override
  void initState() {
    super.initState();
    getSharedText();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(body: Center(child:
Text(dataShared)));
  }

  getSharedText() async {
    var sharedData = await
platform.invokeMethod("getSharedText");
    if (sharedData != null) {
      setState(() {
        dataShared = sharedData;
      });
    }
  }
}
```

What is the equivalent of startActivityForResult()?

The `Navigator` class handles routing in Flutter and is used to get a result back from a route that you have pushed on the stack. This is done by `await`ing on the `Future` returned by `push()`.

For example, to start a location route that lets the user select their location, you could do the following:

```
Map coordinates = await
Navigator.of(context).pushNamed('/location');
```

And then, inside your location route, once the user has selected their location you can `pop` the stack with the result:

```
Navigator.of(context).pop({"lat":43.821757,"long":-
79.226392});
```

Async UI

What is the equivalent of runOnUiThread() in Flutter?

Dart has a single-threaded execution model, with support for `Isolate`s (a way to run Dart code on another thread), an event loop, and asynchronous programming. Unless you spawn an `Isolate`, your Dart code runs in the main UI thread and is driven by an event loop. Flutter's event loop is equivalent to Android's main `Looper`—that is, the `Looper` that is attached to the main thread.

Dart's single-threaded model doesn't mean you need to run everything as a blocking operation that causes the UI to freeze. Unlike Android, which requires you to keep the main thread free at all times, in Flutter, use the asynchronous facilities that the Dart language provides, such as `async`/`await`, to perform asynchronous work. You might be familiar with the `async`/`await` paradigm if you've used it in C#, Javascript, or if you have used Kotlin's coroutines.

For example, you can run network code without causing the UI to hang by using `async`/`await` and letting Dart do the heavy lifting:

```
loadData() async {
  String dataURL =
"https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

Once the `await`ed network call is done, update the UI by calling `setState()`, which triggers a rebuild of the widget sub-tree and updates the data.

The following example loads data asynchronously and displays it in a `ListView`:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

```

```
void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();

    loadData();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView.builder(
          itemCount: widgets.length,
          itemBuilder: (BuildContext context, int
position) {
            return getRow(position);
          }));
  }

  Widget getRow(int i) {
    return Padding(
      padding: EdgeInsets.all(10.0),
      child: Text("Row ${widgets[i]["title"]}")
    );
  }

  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

Refer to the next section for more information on doing work in the background, and how Flutter differs from Android.

How do you move work to a background thread?

In Android, when you want to access a network resource you would typically move to a background thread and do the work, as to not block the main thread, and avoid ANRs. For example, you might be using an `AsyncTask`, a `LiveData`, an `IntentService`, a `JobScheduler` job, or an RxJava pipeline with a scheduler that works on background threads.

Since Flutter is single threaded and runs an event loop (like Node.js), you don't have to worry about thread management or spawning background threads. If you're doing I/O-bound work, such as disk access or a network call, then you can safely use `async`/`await` and you're all set. If, on the other hand, you need to do computationally intensive work that keeps the CPU busy, you want to move it to an `Isolate` to avoid blocking the event loop, like you would keep *any* sort of work out of the main thread in Android.

For I/O-bound work, declare the function as an `async` function, and `await` on long-running tasks inside the function:

```
loadData() async {
  String dataURL =
"https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

This is how you would typically do network or database calls, which are both I/O operations.

On Android, when you extend `AsyncTask`, you typically override 3 methods, `onPreExecute()`, `doInBackground()` and `onPostExecute()`. There is no equivalent in Flutter, since you `await` on a long running function, and Dart's event loop takes care of the rest.

However, there are times when you might be processing a large amount of data and your UI hangs. In Flutter, use `Isolate`s to take advantage of multiple CPU cores to do long-running or computationally intensive tasks.

Isolates are separate execution threads that do not share any memory with the main execution memory heap. This means you can't access variables from the main thread, or update your UI by calling `setState()`. Unlike Android threads, Isolates are true to their name, and cannot share memory (in the form of static fields, for example).

The following example shows, in a simple isolate, how to share data back to the main thread to update the UI.

```
loadData() async {
  ReceivePort receivePort = ReceivePort();
  await Isolate.spawn(dataLoader, receivePort.sendPort);

  // The 'echo' isolate sends its SendPort as the first
message.
  SendPort sendPort = await receivePort.first;

  List msg = await sendReceive(sendPort,
"https://jsonplaceholder.typicode.com/posts");

  setState(() {
    widgets = msg;
  });
}

// The entry point for the isolate.
static dataLoader(SendPort sendPort) async {
  // Open the ReceivePort for incoming messages.
  ReceivePort port = ReceivePort();

  // Notify any other isolates what port this isolate
listens to.
  sendPort.send(port.sendPort);

  await for (var msg in port) {
    String data = msg[0];
    SendPort replyTo = msg[1];

    String dataURL = data;
    http.Response response = await http.get(dataURL);
    // Lots of JSON to parse
    replyTo.send(json.decode(response.body));
  }
}

Future sendReceive(SendPort port, msg) {
  ReceivePort response = ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}
```

Here, `dataLoader()` is the `Isolate` that runs in its own separate execution thread. In the isolate you can perform more CPU intensive processing (parsing a big JSON, for example), or perform computationally intensive math, such as encryption or signal processing.

You can run the full example below:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:async';
import 'dart:isolate';

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    if (widgets.length == 0) {
      return true;
    }

    return false;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(padding: EdgeInsets.all(10.0), child:
Text("Row ${widgets[i]["title"]}"));
  }

  loadData() async {
    ReceivePort receivePort = ReceivePort();
    await Isolate.spawn(dataLoader, receivePort.sendPort);

    // The 'echo' isolate sends its SendPort as the first
message
    SendPort sendPort = await receivePort.first;

    List msg = await sendReceive(sendPort,
"https://jsonplaceholder.typicode.com/posts");

    setState(() {
      widgets = msg;
    });
  }

  // the entry point for the isolate
  static dataLoader(SendPort sendPort) async {
    // Open the ReceivePort for incoming messages.
    ReceivePort port = ReceivePort();

    // Notify any other isolates what port this isolate
listens to.
    sendPort.send(port.sendPort);

    await for (var msg in port) {
      String data = msg[0];
      SendPort replyTo = msg[1];
      String dataURL = data;
      http.Response response = await http.get(dataURL);
      // Lots of JSON to parse
      replyTo.send(json.decode(response.body));
    }
  }

  Future sendReceive(SendPort port, msg) {
    ReceivePort response = ReceivePort();
    port.send([msg, response.sendPort]);
    return response.first;
  }
}
```

What is the equivalent of OkHttp on Flutter?

Making a network call in Flutter is easy when you use the popular `http` package.

While the http package doesn’t have every feature found in OkHttp, it abstracts away much of the networking that you would normally implement yourself, making it a simple way to make network calls.

To use the `http` package, add it to your dependencies in `pubspec.yaml`:

```
dependencies:
  ...
  http: ^0.11.3+16
```

To make a network call, call `await` on the `async` function `http.get()`:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
[...]
  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
```

```
}
```

How do I show the progress for a long-running task?

In Android you would typically show a `ProgressBar` view in your UI while executing a long running task on a background thread.

In Flutter, use a `ProgressIndicator` widget. Show the progress programmatically by controlling when it’s rendered through a boolean flag. Tell Flutter to update its state before your long-running task starts, and hide it after it ends.

In the following example, the build function is separated into three different functions. If `showLoadingDialog()` is `true`(when `widgets.length == 0`), then render the `ProgressIndicator`. Otherwise, render the `ListView` with the data returned from a network call.

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    return widgets.length == 0;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(padding: EdgeInsets.all(10.0), child:
Text("Row ${widgets[i]["title"]}"));
```

```
  }

  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

Project structure & resources

Where do I store my resolution-dependent image files?

While Android treats resources and assets as distinct items, Flutter apps have only assets. All resources that would live in the `res/drawable-*` folders on Android, are placed in an assets folder for Flutter.

Flutter follows a simple density-based format like iOS. Assets might be `1.0x`, `2.0x`, `3.0x`, or any other multiplier. Flutter doesn't have `dp`s but there are logical pixels, which are basically the same as device-independent pixels. The so-called devicePixelRatio expresses the ratio of physical pixels in a single logical pixel.

The equivalent to Android's density buckets are:

Android density qualifier	Flutter pixel ratio
`ldpi`	`0.75x`
`mdpi`	`1.0x`
`hdpi`	`1.5x`
`xhdpi`	`2.0x`
`xxhdpi`	`3.0x`
`xxxhdpi`	`4.0x`

Assets are located in any arbitrary folder—Flutter has no predefined folder structure. You declare the assets (with location) in the `pubspec.yaml` file, and Flutter picks them up.

Note that before Flutter 1.0 beta 2, assets defined in Flutter were not accessible from the native side, and vice versa, native assets and resources weren't available to Flutter, as they lived in separate folders.

As of Flutter beta 2, assets are stored in the native asset folder, and are accessed on the native side using Android's `AssetManager`:

```
val flutterAssetStream =
assetManager.open("flutter_assets/assets/my_flutter_asset.
png")
```

As of Flutter beta 2, Flutter still cannot access native resources, nor it can access native assets.

To add a new image asset called `my_icon.png` to our Flutter project, for example, and deciding that it should live in a folder we arbitrarily called `images`, you would put the base image (1.0x) in the `images` folder, and all the other variants in sub-folders called with the appropriate ratio multiplier:

```
images/my_icon.png       // Base: 1.0x image
images/2.0x/my_icon.png  // 2.0x image
images/3.0x/my_icon.png  // 3.0x image
```

Next, you'll need to declare these images in your `pubspec.yaml` file:

```
assets:
 - images/my_icon.jpeg
```

You can then access your images using `AssetImage`:

```
return AssetImage("images/my_icon.jpeg");
```

or directly in an `Image` widget:

```
@override
Widget build(BuildContext context) {
  return Image.asset("images/my_image.png");
}
```

Where do I store strings? How do I handle localization?

Flutter currently doesn’t have a dedicated resources-like system for strings. At the moment, the best practice is to hold your copy text in a class as static fields and accessing them from there. For example:

```
class Strings {
  static String welcomeMessage = "Welcome To Flutter";
}
```

Then in your code, you can access your strings as such:

```
Text(Strings.welcomeMessage)
```

Flutter has basic support for accessibility on Android, though this feature is a work in progress.

Flutter developers are encouraged to use the intl package for internationalization and localization.

What is the equivalent of a Gradle file? How do I add dependencies?

In Android, you add dependencies by adding to your Gradle build script. Flutter uses Dart’s own build system, and the Pub package manager. The tools delegate the building of the native Android and iOS wrapper apps to the respective build systems.

While there are Gradle files under the `android` folder in your Flutter project, only use these if you are adding native dependencies needed for per-platform integration. In general, use `pubspec.yaml` to declare external dependencies to use in Flutter. A good place to find Flutter packages is Pub.

Activities and fragments

What are the equivalent of activities and fragments in Flutter?

In Android, an `Activity` represents a single focused thing the user can do. A `Fragment` represents a behavior or a portion of user interface. Fragments are a way to modularize your code, compose sophisticated user interfaces for larger screens, and help scale your application UI. In Flutter, both of these concepts fall under the umbrella of `Widget`s.

To learn more about the UI for building Activities and Fragements, see the community-contributed medium article, Flutter For Android Developers : How to design an Activity UI in Flutter.

As mentioned in the Intents section, screens in Flutter are represented by `Widget`s since everything is a widget in Flutter. Use a `Navigator` to move between different `Route`s that represent different screens or pages, or maybe just different states or renderings of the same data.

How do I listen to Android activity lifecycle events?

In Android, you can override methods from the `Activity` to capture lifecycle methods for the activity itself, or register `ActivityLifecycleCallbacks` on the `Application`. In Flutter, you have neither concept, but you can instead listen to lifecycle events by hooking into the `WidgetsBinding` observer and listening to the `didChangeAppLifecycleState()`change event.

The observable lifecycle events are:

- `inactive` — The application is in an inactive state and is not receiving user input. This event only works on iOS, as there is no equivalent event to map to on Android.
- `paused` — The application is not currently visible to the user, not responding to user input, and running in the background. This is equivalent to `onPause()` in Android.
- `resumed` — The application is visible and responding to user input. This is equivalent to `onPostResume()` in Android.
- `suspending` — The application is suspended momentarily. This is equivalent to `onStop` in Android; it is not triggered on iOS as there is no equivalent event to map to on iOS.

For more details on the meaning of these states, see the `AppLifecycleStatus` documentation.

As you might have noticed, only a small minority of the Activity lifecycle events are available; while `FlutterActivity`does capture almost all the activity lifecycle events internally and send them over to the Flutter engine, they're mostly shielded away from you. Flutter takes care of starting and stopping the engine for you, and there is little reason for needing to observe the activity lifecycle on the Flutter side in most cases. If you need to observe the lifecycle to acquire or release any native resources, you should likely be doing it from the native side, at any rate.

Here's an example of how to observe the lifecycle status of the containing activity:

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

class LifecycleWatcher extends StatefulWidget {
  @override
  _LifecycleWatcherState createState() =>
_LifecycleWatcherState();
}

class _LifecycleWatcherState extends
State<LifecycleWatcher> with WidgetsBindingObserver {
  AppLifecycleState _lastLifecycleState;

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
    super.dispose();
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state)
{
    setState(() {
      _lastLifecycleState = state;
    });
  }
  @override
  Widget build(BuildContext context) {
    if (_lastLifecycleState == null)
      return Text('This widget has not observed any
lifecycle changes.', textDirection: TextDirection.ltr);

    return Text('The most recent lifecycle state this
widget observed was: $_lastLifecycleState.',
        textDirection: TextDirection.ltr);
  }
}

void main() {
  runApp(Center(child: LifecycleWatcher()));
}
```

Layouts

What is the equivalent of a LinearLayout?

In Android, a LinearLayout is used to lay your widgets out linearly—either horizontally or vertically. In Flutter, use the Row or Column widgets to achieve the same result.

If you notice the two code samples are identical with the exception of the “Row” and “Column” widget. The children are the same and this feature can be exploited to develop rich layouts that can change overtime with the same children.

```
@override
Widget build(BuildContext context) {
  return Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text('Row One'),
      Text('Row Two'),
      Text('Row Three'),
      Text('Row Four'),
    ],
  );
}
@override
```

```
Widget build(BuildContext context) {
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text('Column One'),
      Text('Column Two'),
      Text('Column Three'),
      Text('Column Four'),
    ],
  );
}
```

To learn more about building linear layouts, see the community contributed medium article Flutter For Android Developers : How to design LinearLayout in Flutter?.

What is the equivalent of a RelativeLayout?

A RelativeLayout lays your widgets out relative to each other. In Flutter, there are a few ways to achieve the same result.

You can achieve the result of a RelativeLayout by using a combination of Column, Row, and Stack widgets. You can specify rules for the widgets constructors on how the children are laid out relative to the parent.

For a good example of building a RelativeLayout in Flutter, see Collin's answer on StackOverflow.

What is the equivalent of a ScrollView?

In Android, use a ScrollView to lay out your widgets—if the user's device has a smaller screen than your content, it scrolls.

In Flutter, the easiest way to do this is using the ListView widget. This might seem like overkill coming from Android, but in Flutter a ListView widget is both a ScrollView and an Android ListView.

```
@override
Widget build(BuildContext context) {
  return ListView(
    children: <Widget>[
      Text('Row One'),
      Text('Row Two'),
      Text('Row Three'),
      Text('Row Four'),
    ],
  );
}
```

How do I handle landscape transitions in Flutter?

FlutterView handles the config change if AndroidManifest.xml contains:

```
android:configChanges="orientation|screenSize"
```

Gesture detection and touch event handling

How do I add an onClick listener to a widget in Flutter?

In Android, you can attach onClick to views such as button by calling the method 'setOnClickListener'.

In Flutter there are two ways of adding touch listeners:

1. If the Widget supports event detection, pass a function to it and handle it in the function. For example, the RaisedButton has an `onPressed` parameter:

```
@override
Widget build(BuildContext context) {
  return RaisedButton(
      onPressed: () {
        print("click");
      },
      child: Text("Button"));
}
```

10. If the Widget doesn't support event detection, wrap the widget in a GestureDetector and pass a function to the `onTap` parameter.

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
      child: GestureDetector(
        child: FlutterLogo(
          size: 200.0,
        ),
        onTap: () {
          print("tap");
        },
      ),
    ));
  }
}
```

How do I handle other gestures on widgets?

Using the GestureDetector, you can listen to a wide range of Gestures such as:

- Tap
  - `onTapDown` - A pointer that might cause a tap has contacted the screen at a particular location.
  - `onTapUp` - A pointer that triggers a tap has stopped contacting the screen at a particular location.
  - `onTap` - A tap has occurred.
  - `onTapCancel` - The pointer that previously triggered the `onTapDown` won't cause a tap.
- Double tap
  - `onDoubleTap` - The user tapped the screen at the same location twice in quick succession.
- Long press
  - `onLongPress` - A pointer has remained in contact with the screen at the same location for a long period of time.
- Vertical drag
  - `onVerticalDragStart` - A pointer has contacted the screen and might begin to move vertically.
  - `onVerticalDragUpdate` - A pointer in contact with the screen has moved further in the vertical direction.
  - `onVerticalDragEnd` - A pointer that was previously in contact with the screen and moving vertically is no longer in contact with the

    screen and was moving at a specific velocity when it stopped contacting the screen.

- Horizontal drag
    - `onHorizontalDragStart` - A pointer has contacted the screen and might begin to move horizontally.
    - `onHorizontalDragUpdate` - A pointer in contact with the screen has moved further in the horizontal direction.
    - `onHorizontalDragEnd` - A pointer that was previously in contact with the screen and moving horizontally is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

The following example shows a `GestureDetector` that rotates the Flutter logo on a double tap:

```
AnimationController controller;
CurvedAnimation curve;

@override
void initState() {
  controller = AnimationController(duration: const
Duration(milliseconds: 2000), vsync: this);
  curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
          child: GestureDetector(
            child: RotationTransition(
                turns: curve,
                child: FlutterLogo(
                  size: 200.0,
                )),
            onDoubleTap: () {
              if (controller.isCompleted) {
                controller.reverse();
              } else {
                controller.forward();
              }
            },
        ),
    ));
  }
}
```

Listviews & adapters

What is the alternative to a ListView in Flutter?

The equivalent to a ListView in Flutter is … a ListView!

In an Android ListView, you create an adapter and pass it into the ListView, which renders each row with what your adapter returns. However, you have to make sure you recycle your rows, otherwise, you get all sorts of crazy visual glitches and memory issues.

Due to Flutter’s immutable widget pattern, you pass a list of widgets to your ListView, and Flutter takes care of making sure that scrolling is fast and smooth.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(Padding(padding: EdgeInsets.all(10.0),
child: Text("Row $i")));
    }
    return widgets;
  }
}
```

How do I know which list item is clicked on?

In Android, the ListView has a method to find out which item was clicked, ‘onItemClickListener’. In Flutter, use the touch handling provided by the passed-in widgets.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(GestureDetector(
        child: Padding(
            padding: EdgeInsets.all(10.0),
            child: Text("Row $i")),
        onTap: () {
          print('row tapped');
        },
      ));
    }
    return widgets;
  }
}
```

How do I update ListView's dynamically?

On Android, you update the adapter and call `notifyDataSetChanged`.

In Flutter, if you were to update the list of widgets inside a `setState()`, you would quickly see that your data did not change visually. This is because when `setState()` is called, the Flutter rendering engine looks at the widget tree to see if anything has changed. When it gets to your `ListView`, it performs a `==` check, and determines that the two `ListView`s are the same. Nothing has changed, so no update is required.

For a simple way to update your `ListView`, create a new `List` inside of `setState()`, and copy the data from the old list to the new list. While this approach is simple, it is not recommended for large data sets, as shown in the next example.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}
```

```
class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = <Widget>[];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: widgets),
    );
  }

  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
          padding: EdgeInsets.all(10.0),
          child: Text("Row $i")),
      onTap: () {
        setState(() {
          widgets = List.from(widgets);
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

The recommended, efficient, and effective way to build a list uses a ListView.Builder. This method is great when you have a dynamic List or a List with very large amounts of data. This is essentially the equivalent of RecyclerView on Android, which automatically recycles list elements for you:

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

void main() {
  runApp(SampleApp());
}
```

```
class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = <Widget>[];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: ListView.builder(
            itemCount: widgets.length,
            itemBuilder: (BuildContext context, int
position) {
              return getRow(position);
            }));
  }
  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
          padding: EdgeInsets.all(10.0),
          child: Text("Row $i")),
      onTap: () {
        setState(() {
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

Instead of creating a “ListView”, create a ListView.builder that takes two key parameters: the initial length of the list, and an ItemBuilder function.

The ItemBuilder function is similar to the `getView` function in an Android adapter; it takes a position, and returns the row you want rendered at that position.

Finally, but most importantly, notice that the `onTap()` function doesn’t recreate the list anymore, but instead `.add`s to it.

Working with text

How do I set custom fonts on my Text widgets?

In Android SDK (as of Android O), you create a Font resource file and pass it into the FontFamily param for your TextView.

In Flutter, place the font file in a folder and reference it in the `pubspec.yaml` file, similar to how you import images.

```
fonts:
   - family: MyCustomFont
     fonts:
       - asset: fonts/MyCustomFont.ttf
       - style: italic
```

Then assign the font to your `Text` widget:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: Text(
        'This is a custom font text',
        style: TextStyle(fontFamily: 'MyCustomFont'),
      ),
    ),
  );
}
```

How do I style my Text widgets?

Along with fonts, you can customize other styling elements on a `Text` widget. The style parameter of a `Text` widget takes a `TextStyle` object, where you can customize many parameters, such as:

- color
- decoration
- decorationColor
- decorationStyle
- fontFamily
- fontSize
- fontStyle
- fontWeight
- hashCode
- height
- inherit
- letterSpacing
- textBaseline
- wordSpacing

Form input

For more information on using Forms, see Récupérer la valeur d'un champ texte , from the Flutter Cookbook.

What is the equivalent of a “hint” on an Input?

In Flutter, you can easily show a “hint” or a placeholder text for your input by adding an InputDecoration object to the decoration constructor parameter for the Text Widget.

```
body: Center(
  child: TextField(
    decoration: InputDecoration(hintText: "This is a
hint"),
  )
)
```

How do I show validation errors?

Just as you would with a “hint”, pass an InputDecoration object to the decoration constructor for the Text widget.

However, you don’t want to start off by showing an error. Instead, when the user has entered invalid data, update the state, and pass a new `InputDecoration` object.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  String _errorText;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: TextField(
          onSubmitted: (String text) {
            setState(() {
              if (!isEmail(text)) {
                _errorText = 'Error: This is not an
email';
              } else {
                _errorText = null;
              }
            });
          },
          decoration: InputDecoration(hintText: "This is a
hint", errorText: _getErrorText()),
        ),
      ),
    );
  }

  _getErrorText() {
    return _errorText;
  }

  bool isEmail(String em) {
    String emailRegexp =

r'^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\
".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-
9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

    RegExp regExp = RegExp(emailRegexp);
    return regExp.hasMatch(em);
  }
}
```

Flutter plugins

How do I access the GPS sensor?

Use the `geolocator` community plugin.

How do I access the camera?

The `image_picker` plugin is popular for accessing the camera.

How do I log in with Facebook?

To Log in with Facebook, use the `flutter_facebook_login` community plugin.

How do I use Firebase features?

Most Firebase functions are covered by first party plugins. These plugins are first-party integrations, maintained by the Flutter team:

- `firebase_admob` for Firebase AdMob
- `firebase_analytics` for Firebase Analytics
- `firebase_auth` for Firebase Auth
- `firebase_database` for Firebase RTDB
- `firebase_storage` for Firebase Cloud Storage
- `firebase_messaging` for Firebase Messaging (FCM)
- `flutter_firebase_ui` for quick Firebase Auth integrations (Facebook, Google, Twitter and email)
- `cloud_firestore` for Firebase Cloud Firestore

You can also find some third-party Firebase plugins on Pub that cover areas not directly covered by the first-party plugins.

How do I build my own custom native integrations?

If there is platform-specific functionality that Flutter or its community Plugins are missing, you can build your own following the developing packages and plugins page.

Flutter’s plugin architecture, in a nutshell, is much like using an Event bus in Android: you fire off a message and let the receiver process and emit a result back to you. In this case, the receiver is code running on the native side on Android or iOS.

How do I use the NDK in my Flutter application?

If you use the NDK in your current Android application and want your Flutter application to take advantage of your native libraries then it’s possible by building a custom plugin.

Your custom plugin first talks to your Android app, where you call your `native` functions over JNI. Once a response is ready, send a message back to Flutter and render the result.

Calling native code directly from Flutter is currently not supported.

Themes

How do I theme my app?

Out of the box, Flutter comes with a beautiful implementation of Material Design, which takes care of a lot of styling and theming needs that you would typically do. Unlike Android where you declare themes in XML and then assign it to your application using AndroidManifest.xml, in Flutter you declare themes in the top level widget.

To take full advantage of Material Components in your app, you can declare a top level widget `MaterialApp` as the entry point to your application. MaterialApp is a convenience widget that wraps a number of widgets that are commonly required for applications implementing Material Design. It builds upon a WidgetsApp by adding Material specific functionality.

You can also use a `WidgetApp` as your app widget, which provides some of the same functionality, but is not as rich as `MaterialApp`.

To customize the colors and styles of any child components, pass a `ThemeData` object to the `MaterialApp` widget. For example, in the code below, the primary swatch is set to blue and text selection color is red.

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textSelectionColor: Colors.red
      ),
      home: SampleAppPage(),
    );
  }
}
```

Databases and local storage

How do I access Shared Preferences?

In Android, you can store a small collection of key-value pairs using the SharedPreferences API.

In Flutter, access this functionality using the Shared_Preferences plugin. This plugin wraps the functionality of both Shared Preferences and NSUserDefaults (the iOS equivalent).

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

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body: Center(
          child: RaisedButton(
            onPressed: _incrementCounter,
            child: Text('Increment Counter'),
          ),
        ),
      ),
    ),
  );
}

_incrementCounter() async {
  SharedPreferences prefs = await
SharedPreferences.getInstance();
  int counter = (prefs.getInt('counter') ?? 0) + 1;
  print('Pressed $counter times.');
  prefs.setInt('counter', counter);
}
```

How do I access SQLite in Flutter?

In Android, you use SQLite to store structured data that you can query using SQL.

In Flutter, access this functionality using the SQFlite plugin.

Debugging

What tools can I use to debug my app in Flutter?

Use the DevTools suite for debugging Flutter or Dart apps.

DevTools includes support for profiling, examining the heap, inspecting the widget tree, logging diagnostics, debugging, observing executed lines of code, debugging memory leaks and memory fragmentation. For more information, see theDevTools documentation.

Notifications

How do I set up push notifications?

In Android, you use Firebase Cloud Messaging to setup push notifications for your app.

In Flutter, access this functionality using the Firebase Messaging plugin. For more information on using the Firebase Cloud Messaging API, see the [`firebase_messaging`][] plugin documentation.

Flutter pour les développeurs iOS

This document is for iOS developers looking to apply their existing iOS knowledge to build mobile apps with Flutter. If you understand the fundamentals of the iOS framework then you can use this document as a way to get started learning Flutter development.

Before diving into this doc, you might want to watch a 15-minute video from the Flutter Youtube channel about the Cupertino package.

Your iOS knowledge and skill set are highly valuable when building with Flutter, because Flutter relies on the mobile operating system for numerous capabilities and configurations. Flutter is a new way to build UIs for mobile, but it has a plugin system to communicate with iOS (and Android) for non-UI tasks. If you're an expert in iOS development, you don't have to relearn everything to use Flutter.

Flutter also already makes a number of adaptations in the framework for you when running on iOS. For a list, see Platform adaptations.

This document can be used as a cookbook by jumping around and finding questions that are most relevant to your needs.

Views

What is the equivalent of a UIView in Flutter?

How is react-style, or *declarative*, programming different than the traditional imperative style? For a comparison, see Introduction to declarative UI.

On iOS, most of what you create in the UI is done using view objects, which are instances of the `UIView` class. These can act as containers for other `UIView` classes, which form your layout.

In Flutter, the rough equivalent to a `UIView` is a `Widget`. Widgets don't map exactly to iOS views, but while you're getting acquainted with how Flutter works you can think of them as "the way you declare and construct UI".

However, these have a few differences to a `UIView`. To start, widgets have a different lifespan: they are immutable and only exist until they need to be changed. Whenever widgets or their state change, Flutter's framework creates a new tree of widget instances. In comparison, an iOS view is not recreated when it changes, but rather it's a mutable entity that is drawn once and doesn't redraw until it is invalidated using `setNeedsDisplay()`.

Furthermore, unlike `UIView`, Flutter's widgets are lightweight, in part due to their immutability. Because they aren't views themselves, and aren't directly drawing anything, but rather are a description of the UI and its semantics that get "inflated" into actual view objects under the hood.

Flutter includes the Material Components library. These are widgets that implement the Material Design guidelines. Material Design is a flexible design system optimized for all platforms, including iOS.

But Flutter is flexible and expressive enough to implement any design language. On iOS, you can use the Cupertino widgets to produce an interface that looks like Apple's iOS design language.

How do I update widgets?

To update your views on iOS, you directly mutate them. In Flutter, widgets are immutable and not updated directly. Instead, you have to manipulate the widget's state.

This is where the concept of Stateful vs Stateless widgets comes in. A `StatelessWidget` is just what it sounds like—a widget with no state attached.

`StatelessWidgets` are useful when the part of the user interface you are describing does not depend on anything other than the initial configuration information in the widget.

For example, in iOS, this is similar to placing a `UIImageView` with your logo as the `image`. If the logo is not changing during runtime, use a `StatelessWidget` in Flutter.

If you want to dynamically change the UI based on data received after making an HTTP call, use a `StatefulWidget`. After the HTTP call has completed, tell the Flutter framework that the widget's `State` is updated, so it can update the UI.

The important difference between stateless and stateful widgets is that `StatefulWidget`s have a `State` object that stores state data and carries it over across tree rebuilds, so it's not lost.

If you are in doubt, remember this rule: if a widget changes outside of the `build` method (because of runtime user interactions, for example), it's stateful. If the widget never changes, once built, it's stateless. However, even if a widget is stateful, the containing parent widget can still be stateless if it isn't itself reacting to those changes (or other inputs).

The following example shows how to use a `StatelessWidget`. A common `StatelessWidget` is the `Text` widget. If you look at the implementation of the `Text` widget you'll find it subclasses `StatelessWidget`.

```
Text(
  'I like Flutter!',
  style: TextStyle(fontWeight: FontWeight.bold),
);
```

If you look at the code above, you might notice that the `Text` widget carries no explicit state with it. It renders what is passed in its constructors and nothing more.

But, what if you want to make "I Like Flutter" change dynamically, for example when clicking a `FloatingActionButton`?

To achieve this, wrap the `Text` widget in a `StatefulWidget` and update it when the user clicks the button.

For example:

```
class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  // Default placeholder text
  String textToShow = "I Like Flutter";
  void _updateText() {
    setState(() {
      // update the text
      textToShow = "Flutter is Awesome!";
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(child: Text(textToShow)),
      floatingActionButton: FloatingActionButton(
        onPressed: _updateText,
        tooltip: 'Update Text',
        child: Icon(Icons.update),
      ),
    );
  }
}
```

How do I lay out my widgets? Where is my Storyboard?

In iOS, you might use a Storyboard file to organize your views and set constraints, or you might set your constraints programmatically in your view controllers. In Flutter, declare your layout in code by composing a widget tree.

The following example shows how to display a simple widget with padding:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: CupertinoButton(
        onPressed: () {
          setState(() { _pressedCount += 1; });
        },
        child: Text('Hello'),
        padding: EdgeInsets.only(left: 10.0, right: 10.0),
      ),
    ),
  );
}
```

You can add padding to any widget, which mimics the functionality of constraints in iOS.

You can view the layouts that Flutter has to offer in the widget catalog.

How do I add or remove a component from my layout?

In iOS, you call `addSubview()` on the parent, or `removeFromSuperview()` on a child view to dynamically add or remove child views. In Flutter, because widgets are immutable there is no direct equivalent to `addSubview()`. Instead, you can pass a function to the parent that returns a widget, and control that child's creation with a boolean flag.

The following example shows how to toggle between two widgets when the user clicks the `FloatingActionButton`:

```
class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  // Default value for toggle
  bool toggle = true;
  void _toggle() {
    setState(() {
      toggle = !toggle;
    });
  }

  _getToggleChild() {
    if (toggle) {
      return Text('Toggle One');
    } else {
      return CupertinoButton(
        onPressed: () {},
        child: Text('Toggle Two'),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: _getToggleChild(),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _toggle,
        tooltip: 'Update Text',
        child: Icon(Icons.update),
      ),
    );
  }
}
```

How do I animate a widget?

In iOS, you create an animation by calling the `animate(withDuration:animations:)` method on a view. In Flutter, use the animation library to wrap widgets inside an animated widget.

In Flutter, use an `AnimationController`, which is an `Animation<double>` that can pause, seek, stop, and reverse the animation. It requires a `Ticker` that signals when vsync happens and produces a linear interpolation between 0 and 1 on each frame while it's running. You then create one or more `Animation`s and attach them to the controller.

For example, you might use `CurvedAnimation` to implement an animation along an interpolated curve. In this sense, the controller is the "master" source of the animation progress and the `CurvedAnimation` computes the curve that replaces the controller's default linear motion. Like widgets, animations in Flutter work with composition.

When building the widget tree you assign the `Animation` to an animated property of a widget, such as the opacity of a `FadeTransition`, and tell the controller to start the animation.

The following example shows how to write a `FadeTransition` that fades the widget into a logo when you press the `FloatingActionButton`:

```
class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Fade Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyFadeTest(title: 'Fade Demo'),
    );
  }
}

class MyFadeTest extends StatefulWidget {
  MyFadeTest({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyFadeTest createState() => _MyFadeTest();
}

class _MyFadeTest extends State<MyFadeTest> with
TickerProviderStateMixin {
  AnimationController controller;
  CurvedAnimation curve;

  @override
  void initState() {
    controller = AnimationController(duration: const
Duration(milliseconds: 2000), vsync: this);
    curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Container(
          child: FadeTransition(
            opacity: curve,
            child: FlutterLogo(
              size: 100.0,
            )
          )
        )
      ),
      floatingActionButton: FloatingActionButton(
        tooltip: 'Fade',
        child: Icon(Icons.brush),
        onPressed: () {
          controller.forward();
        },
      ),
    );
  }

  @override
  dispose() {
    controller.dispose();
    super.dispose();
  }
}
```

For more information, see Animation & Motion widgets, the Animations tutorial, and the Animations overview.

How do I draw to the screen?

On iOS, you use `CoreGraphics` to draw lines and shapes to the screen. Flutter has a different API based on the `Canvas`class, with two other classes that help you draw: `CustomPaint` and `CustomPainter`, the latter of which implements your algorithm to draw to the canvas.

To learn how to implement a signature painter in Flutter, see Collin's answer on StackOverflow.

```
class SignaturePainter extends CustomPainter {
  SignaturePainter(this.points);

  final List<Offset> points;

  void paint(Canvas canvas, Size size) {
    var paint = Paint()
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5.0;
    for (int i = 0; i < points.length - 1; i++) {
      if (points[i] != null && points[i + 1] != null)
        canvas.drawLine(points[i], points[i + 1], paint);
    }
  }

  bool shouldRepaint(SignaturePainter other) =>
other.points != points;
}

class Signature extends StatefulWidget {
  SignatureState createState() => SignatureState();
}

class SignatureState extends State<Signature> {

  List<Offset> _points = <Offset>[];

  Widget build(BuildContext context) {
    return GestureDetector(
      onPanUpdate: (DragUpdateDetails details) {
        setState(() {
          RenderBox referenceBox =
context.findRenderObject();
          Offset localPosition =

referenceBox.globalToLocal(details.globalPosition);
          _points =
List.from(_points)..add(localPosition);
        });
      },
      onPanEnd: (DragEndDetails details) =>
_points.add(null),
      child: CustomPaint(painter:
SignaturePainter(_points), size: Size.infinite),
    );
  }
}
```

Where is the widget’s opacity?

On iOS, everything has .opacity or .alpha. In Flutter, most of the time you need to wrap a widget in an Opacity widget to accomplish this.

How do I build custom widgets?

In iOS, you typically subclass `UIView`, or use a pre-existing view, to override and implement methods that achieve the desired behavior. In Flutter, build a custom widget by composing smaller widgets (instead of extending them).

For example, how do you build a `CustomButton` that takes a label in the constructor? Create a CustomButton that composes a `RaisedButton` with a label, rather than by extending `RaisedButton`:

```
class CustomButton extends StatelessWidget {
  final String label;

  CustomButton(this.label);

  @override
  Widget build(BuildContext context) {
    return RaisedButton(onPressed: () {}, child:
Text(label));
  }
}
```

Then use `CustomButton`, just as you’d use any other Flutter widget:

```
@override
Widget build(BuildContext context) {
  return Center(
    child: CustomButton("Hello"),
  );
}
```

Navigation

How do I navigate between pages?

In iOS, to travel between view controllers, you can use a `UINavigationController` that manages the stack of view controllers to display.

Flutter has a similar implementation, using a `Navigator` and `Routes`. A `Route` is an abstraction for a “screen” or “page” of an app, and a `Navigator` is a widget that manages routes. A route roughly maps to a `UIViewController`. The navigator works in a similar way to the iOS `UINavigationController`, in that it can `push()` and `pop()` routes depending on whether you want to navigate to, or back from, a view.

To navigate between pages, you have a couple options:

- Specify a `Map` of route names.
- Directly navigate to a route.

The following example builds a Map.

```
void main() {
  runApp(CupertinoApp(
    home: MyAppHome(), // becomes the route named '/'
    routes: <String, WidgetBuilder> {
      '/a': (BuildContext context) => MyPage(title: 'page
A'),
      '/b': (BuildContext context) => MyPage(title: 'page
B'),
      '/c': (BuildContext context) => MyPage(title: 'page
C'),
    },
  ));
}
```

Navigate to a route by `push`ing its name to the `Navigator`.

```
Navigator.of(context).pushNamed('/b');
```

The `Navigator` class handles routing in Flutter and is used to get a result back from a route that you have pushed on the stack. This is done by `await`ing on the `Future` returned by `push()`.

For example, to start a 'location' route that lets the user select their location, you might do the following:

```
Map coordinates = await
Navigator.of(context).pushNamed('/location');
```

And then, inside your 'location' route, once the user has selected their location, `pop()` the stack with the result:

```
Navigator.of(context).pop({"lat":43.821757,"long":-
79.226392});
```

How do I navigate to another app?

In iOS, to send the user to another application, you use a specific URL scheme. For the system level apps, the scheme depends on the app. To implement this functionality in Flutter, create a native platform integration, or use an existing plugin, such as `url_launcher`.

How do I pop back to the iOS native viewcontroller?

Calling `SystemNavigator.pop()` from your Dart code invokes the following iOS code:

```
UIViewController* viewController = [UIApplication
sharedApplication].keyWindow.rootViewController;
  if ([viewController
isKindOfClass:[UINavigationController class]]) {
    [((UINavigationController*)viewController)
popViewControllerAnimated:NO];
  }
```

If that doesn't do what you want, you can create your own platform channel to invoke arbitrary iOS code.

Threading & asynchronicity

How do I write asynchronous code?

Dart has a single-threaded execution model, with support for `Isolate`s (a way to run Dart code on another thread), an event loop, and asynchronous programming. Unless you spawn an `Isolate`, your Dart code runs in the main UI thread and is driven by an event loop. Flutter's event loop is equivalent to the iOS main loop—that is, the `Looper` that is attached to the main thread.

Dart's single-threaded model doesn't mean you are required to run everything as a blocking operation that causes the UI to freeze. Instead, use the asynchronous facilities that the Dart language provides, such as `async`/`await`, to perform asynchronous work.

For example, you can run network code without causing the UI to hang by using `async`/`await` and letting Dart do the heavy lifting:

```
loadData() async {
  String dataURL =
"https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

Once the `await`ed network call is done, update the UI by calling `setState()`, which triggers a rebuild of the widget sub-tree and updates the data.

The following example loads data asynchronously and displays it in a `ListView`:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();

    loadData();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView.builder(
          itemCount: widgets.length,
          itemBuilder: (BuildContext context, int
position) {
            return getRow(position);
          }));
  }

  Widget getRow(int i) {
    return Padding(
      padding: EdgeInsets.all(10.0),
      child: Text("Row ${widgets[i]["title"]}")
    );
  }

  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

Refer to the next section for more information on doing work in the background, and how Flutter differs from iOS.

How do you move work to a background thread?

Since Flutter is single threaded and runs an event loop (like Node.js), you don’t have to worry about thread management or spawning background threads. If you’re doing I/O-bound work, such as disk access or a network call, then you can safely use `async`/`await` and you’re done. If, on the other hand, you need to do computationally intensive work that keeps the CPU busy, you want to move it to an `Isolate` to avoid blocking the event loop.

For I/O-bound work, declare the function as an `async` function, and `await` on long-running tasks inside the function:

```
loadData() async {
  String dataURL =
"https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = json.decode(response.body);
  });
}
```

This is how you typically do network or database calls, which are both I/O operations.

However, there are times when you might be processing a large amount of data and your UI hangs. In Flutter, use `Isolate`s to take advantage of multiple CPU cores to do long-running or computationally intensive tasks.

Isolates are separate execution threads that do not share any memory with the main execution memory heap. This means you can’t access variables from the main thread, or update your UI by calling `setState()`. Isolates are true to their name, and cannot share memory (in the form of static fields, for example).

The following example shows, in a simple isolate, how to share data back to the main thread to update the UI.

```
loadData() async {
  ReceivePort receivePort = ReceivePort();
  await Isolate.spawn(dataLoader, receivePort.sendPort);

  // The 'echo' isolate sends its SendPort as the first
message
  SendPort sendPort = await receivePort.first;

  List msg = await sendReceive(sendPort,
"https://jsonplaceholder.typicode.com/posts");

  setState(() {
    widgets = msg;
  });
}

// The entry point for the isolate
static dataLoader(SendPort sendPort) async {
  // Open the ReceivePort for incoming messages.
  ReceivePort port = ReceivePort();

  // Notify any other isolates what port this isolate
listens to.
  sendPort.send(port.sendPort);

  await for (var msg in port) {
    String data = msg[0];
    SendPort replyTo = msg[1];

    String dataURL = data;
    http.Response response = await http.get(dataURL);
    // Lots of JSON to parse
    replyTo.send(json.decode(response.body));
  }
}

Future sendReceive(SendPort port, msg) {
  ReceivePort response = ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}
```

Here, `dataLoader()` is the `Isolate` that runs in its own separate execution thread. In the isolate you can perform more CPU intensive processing (parsing a big JSON, for example), or perform computationally intensive math, such as encryption or signal processing.

You can run the full example below:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:async';
import 'dart:isolate';

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }
  showLoadingDialog() {
    if (widgets.length == 0) {
      return true;
    }

    return false;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(padding: EdgeInsets.all(10.0), child:
Text("Row ${widgets[i]["title"]}"));
  }

  loadData() async {
    ReceivePort receivePort = ReceivePort();
    await Isolate.spawn(dataLoader, receivePort.sendPort);

    // The 'echo' isolate sends its SendPort as the first
message
```

```
    SendPort sendPort = await receivePort.first;

    List msg = await sendReceive(sendPort,
"https://jsonplaceholder.typicode.com/posts");

    setState(() {
      widgets = msg;
    });
  }

// the entry point for the isolate
  static dataLoader(SendPort sendPort) async {
    // Open the ReceivePort for incoming messages.
    ReceivePort port = ReceivePort();

    // Notify any other isolates what port this isolate
listens to.
    sendPort.send(port.sendPort);

    await for (var msg in port) {
      String data = msg[0];
      SendPort replyTo = msg[1];

      String dataURL = data;
      http.Response response = await http.get(dataURL);
      // Lots of JSON to parse
      replyTo.send(json.decode(response.body));
    }
  }

  Future sendReceive(SendPort port, msg) {
    ReceivePort response = ReceivePort();
    port.send([msg, response.sendPort]);
    return response.first;
  }
}
```

How do I make network requests?

Making a network call in Flutter is easy when you use the popular `http` package. This abstracts away a lot of the networking that you might normally implement yourself, making it simple to make network calls.

To use the `http` package, add it to your dependencies in `pubspec.yaml`:

```
dependencies:
  ...
  http: ^0.11.3+16
```

To make a network call, call `await` on the `async` function `http.get()`:

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
[...]
  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

How do I show the progress of a long-running task?

In iOS, you typically use a `UIProgressView` while executing a long-running task in the background.

In Flutter, use a `ProgressIndicator` widget. Show the progress programmatically by controlling when it's rendered through a boolean flag. Tell Flutter to update its state before your long-running task starts, and hide it after it ends.

In the example below, the build function is separated into three different functions. If `showLoadingDialog()` is `true` (when `widgets.length == 0`), then render the `ProgressIndicator`. Otherwise, render the `ListView` with the data returned from a network call.

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(SampleApp());
}

```

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    return widgets.length == 0;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return Center(child: CircularProgressIndicator());
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Sample App"),
        ),
        body: getBody());
  }

  ListView getListView() => ListView.builder(
      itemCount: widgets.length,
      itemBuilder: (BuildContext context, int position) {
        return getRow(position);
      });

  Widget getRow(int i) {
    return Padding(padding: EdgeInsets.all(10.0), child:
Text("Row ${widgets[i]["title"]}"));
  }

  loadData() async {
    String dataURL =
"https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
      widgets = json.decode(response.body);
    });
  }
}
```

Project structure, localization, dependencies and assets

How do I include image assets for Flutter? What about multiple resolutions?

While iOS treats images and assets as distinct items, Flutter apps have only assets. Resources that are placed in the `Images.xcasset` folder on iOS, are placed in an assets folder for Flutter. As with iOS, assets are any type of file, not just images. For example, you might have a JSON file located in the `my-assets` folder:

```
my-assets/data.json
```

Declare the asset in the `pubspec.yaml` file:

```
assets:
 - my-assets/data.json
```

And then access it from code using an `AssetBundle`:

```
import 'dart:async' show Future;
import 'package:flutter/services.dart' show rootBundle;

Future<String> loadAsset() async {
  return await rootBundle.loadString('my-
assets/data.json');
}
```

For images, Flutter follows a simple density-based format like iOS. Image assets might be `1.0x`, `2.0x`, `3.0x`, or any other multiplier. The so-called `devicePixelRatio` expresses the ratio of physical pixels in a single logical pixel.

Assets are located in any arbitrary folder—Flutter has no predefined folder structure. You declare the assets (with location) in the `pubspec.yaml` file, and Flutter picks them up.

For example, to add an image called `my_icon.png` to your Flutter project, you might decide to store it in a folder arbitrarily called `images`. Place the base image (1.0x) in the `images` folder, and the other variants in sub-folders named after the appropriate ratio multiplier:

```
images/my_icon.png        // Base: 1.0x image
images/2.0x/my_icon.png   // 2.0x image
images/3.0x/my_icon.png   // 3.0x image
```

Next, declare these images in the `pubspec.yaml` file:

```
assets:
 - images/my_icon.png
```

You can now access your images using `AssetImage`:

```
return AssetImage("images/a_dot_burr.jpeg");
```

or directly in an `Image` widget:

```
@override
```

```
Widget build(BuildContext context) {
  return Image.asset("images/my_image.png");
}
```

For more details, see Adding Assets and Images in Flutter.

Where do I store strings? How do I handle localization?

Unlike iOS, which has the `Localizable.strings` file, Flutter doesn't currently have a dedicated system for handling strings. At the moment, the best practice is to declare your copy text in a class as static fields and access them from there. For example:

```
class Strings {
  static String welcomeMessage = "Welcome To Flutter";
}
```

You can access your strings as such:

```
Text(Strings.welcomeMessage)
```

By default, Flutter only supports US English for its strings. If you need to add support for other languages, include the `flutter_localizations` package. You might also need to add Dart's `intl` package to use i10n machinery, such as date/time formatting.

```
dependencies:
  # ...
  flutter_localizations:
    sdk: flutter
  intl: "^0.15.6"
```

To use the `flutter_localizations` package, specify the `localizationsDelegates` and `supportedLocales` on the app widget:

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

MaterialApp(
 localizationsDelegates: [
   // Add app-specific localization delegate[s] here
   GlobalMaterialLocalizations.delegate,
   GlobalWidgetsLocalizations.delegate,
 ],
 supportedLocales: [
    const Locale('en', 'US'), // English
    const Locale('he', 'IL'), // Hebrew
    // ... other locales the app supports
  ],
  // ...
)
```

The delegates contain the actual localized values, while the `supportedLocales` defines which locales the app supports. The above example uses a `MaterialApp`, so it has both a `GlobalWidgetsLocalizations` for the base widgets localized values, and a `MaterialWidgetsLocalizations` for the Material widgets localizations. If you use `WidgetsApp` for your app, you don't need the latter. Note that these two delegates contain "default" values, but you'll need to provide one or more delegates for your own app's localizable copy, if you want those to be localized too.

When initialized, the `WidgetsApp` (or `MaterialApp`) creates a `Localizations` widget for you, with the delegates you specify. The current locale for the device is always accessible from the `Localizations` widget from the current context (in the form of a `Locale` object), or using the `Window.locale`.

To access localized resources, use the `Localizations.of()` method to access a specific localizations class that is provided by a given delegate. Use the `intl_translation` package to extract translatable copy to arb files for translating, and importing them back into the app for using them with `intl`.

For further details on internationalization and localization in Flutter, see the internationalization guide, which has sample code with and without the `intl` package.

Note that before Flutter 1.0 beta 2, assets defined in Flutter were not accessible from the native side, and vice versa, native assets and resources weren't available to Flutter, as they lived in separate folders.

What is the equivalent of CocoaPods? How do I add dependencies?

In iOS, you add dependencies by adding to your `Podfile`. Flutter uses Dart's build system and the Pub package manager to handle dependencies. The tools delegate the building of the native Android and iOS wrapper apps to the respective build systems.

While there is a Podfile in the iOS folder in your Flutter project, only use this if you are adding native dependencies needed for per-platform integration. In general, use `pubspec.yaml` to declare external dependencies in Flutter. A good place to find great packages for Flutter is the Pub site.

ViewControllers

What is the equivalent to ViewController in Flutter?

In iOS, a `ViewController` represents a portion of user interface, most commonly used for a screen or section. These are composed together to build complex user interfaces, and help scale your application's UI. In Flutter, this job falls to Widgets. As mentioned in the Navigation section, screens in Flutter are represented by Widgets since "everything is a widget!" Use a `Navigator` to move between different `Route`s that represent different screens or pages, or maybe different states or renderings of the same data.

How do I listen to iOS lifecycle events?

In iOS, you can override methods to the `ViewController` to capture lifecycle methods for the view itself, or register lifecycle callbacks in the `AppDelegate`. In Flutter you have neither concept, but you can instead listen to lifecycle events by hooking into the `WidgetsBinding` observer and listening to the `didChangeAppLifecycleState()` change event.

The observable lifecycle events are:

- `inactive` — The application is in an inactive state and is not receiving user input. This event only works on iOS, as there is no equivalent event on Android.
- `paused` — The application is not currently visible to the user, is not responding to user input, but is running in the background.
- `resumed` — The application is visible and responding to user input.
- `suspending` — The application is suspended momentarily. The iOS platform has no equivalent event.

For more details on the meaning of these states, see `AppLifecycleStatus` documentation.

Layouts

What is the equivalent of UITableView or UICollectionView in Flutter?

In iOS, you might show a list in either a `UITableView` or a `UICollectionView`. In Flutter, you have a similar implementation using a `ListView`. In iOS, these views have delegate methods for deciding the number of rows, the cell for each index path, and the size of the cells.

Due to Flutter's immutable widget pattern, you pass a list of widgets to your `ListView`, and Flutter takes care of making sure that scrolling is fast and smooth.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(Padding(padding: EdgeInsets.all(10.0), 
child: Text("Row $i")));
    }
    return widgets;
  }
}
```

How do I know which list item is clicked?

In iOS, you implement the delegate method, `tableView:didSelectRowAtIndexPath:`. In Flutter, use the touch handling provided by the passed-in widgets.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(GestureDetector(
        child: Padding(
          padding: EdgeInsets.all(10.0),
          child: Text("Row $i"),
        ),
        onTap: () {
          print('row tapped');
        },
      ));
    }
    return widgets;
  }
}
```

How do I dynamically update a ListView?

In iOS, you update the data for the list view, and notify the table or collection view using the `reloadData` method.

In Flutter, if you update the list of widgets inside a `setState()`, you quickly see that your data doesn't change visually. This is because when `setState()` is called, the Flutter rendering engine looks at the widget tree to see if anything has changed. When it gets to your `ListView`, it performs an `==` check, and determines that the two `ListView`s are the same. Nothing has changed, so no update is required.

For a simple way to update your `ListView`, create a new `List` inside of `setState()`, and copy the data from the old list to the new list. While this approach is simple, it is not recommended for large data sets, as shown in the next example.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView(children: widgets),
    );
  }

  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row $i"),
      ),
      onTap: () {
        setState(() {
          widgets = List.from(widgets);
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

The recommended, efficient, and effective way to build a list uses a `ListView.Builder`. This method is great when you have a dynamic list or a list with very large amounts of data.

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

void main() {
  runApp(SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: ListView.builder(
        itemCount: widgets.length,
        itemBuilder: (BuildContext context, int position)
{
          return getRow(position);
        },
      ),
    );
  }

  Widget getRow(int i) {
    return GestureDetector(
      child: Padding(
        padding: EdgeInsets.all(10.0),
        child: Text("Row $i"),
      ),
      onTap: () {
        setState(() {
          widgets.add(getRow(widgets.length + 1));
          print('row $i');
        });
      },
    );
  }
}
```

Instead of creating a “ListView”, create a `ListView.builder` that takes two key parameters: the initial length of the list, and an `ItemBuilder` function.

The `ItemBuilder` function is similar to the `cellForItemAt` delegate method in an iOS table or collection view, as it takes a position, and returns the cell you want rendered at that position.

Finally, but most importantly, notice that the `onTap()` function doesn’t recreate the list anymore, but instead `.add`s to it.

What is the equivalent of a ScrollView in Flutter?

In iOS, you wrap your views in a `ScrollView` that allows a user to scroll your content if needed.

In Flutter the easiest way to do this is using the `ListView` widget. This acts as both a `ScrollView` and an iOS `TableView`, as you can layout widgets in a vertical format.

```
@override
Widget build(BuildContext context) {
  return ListView(
    children: <Widget>[
      Text('Row One'),
      Text('Row Two'),
      Text('Row Three'),
      Text('Row Four'),
    ],
  );
}
```

For more detailed docs on how to lay out widgets in Flutter, see the layout tutorial.

Gesture detection and touch event handling

How do I add a click listener to a widget in Flutter?

In iOS, you attach a `GestureRecognizer` to a view to handle click events. In Flutter, there are two ways of adding touch listeners:

1. If the widget supports event detection, pass a function to it, and handle the event in the function. For example, the`RaisedButton` widget has an `onPressed` parameter:

```
@override
Widget build(BuildContext context) {
  return RaisedButton(
    onPressed: () {
      print("click");
    },
    child: Text("Button"),
  );
}
```

11. If the Widget doesn’t support event detection, wrap the widget in a GestureDetector and pass a function to the `onTap` parameter.

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: GestureDetector(
          child: FlutterLogo(
            size: 200.0,
          ),
          onTap: () {
            print("tap");
          },
        ),
      ),
    );
  }
}
```

How do I handle other gestures on widgets?

Using `GestureDetector` you can listen to a wide range of gestures such as:

- Tapping
  - `onTapDown` — A pointer that might cause a tap has contacted the screen at a particular location.
  - `onTapUp` — A pointer that triggers a tap has stopped contacting the screen at a particular location.
  - `onTap` — A tap has occurred.
  - `onTapCancel` — The pointer that previously triggered the `onTapDown` won't cause a tap.
- Double tapping
  - `onDoubleTap` — The user tapped the screen at the same location twice in quick succession.
- Long pressing
  - `onLongPress` — A pointer has remained in contact with the screen at the same location for a long period of time.
- Vertical dragging
  - `onVerticalDragStart` — A pointer has contacted the screen and might begin to move vertically.
  - `onVerticalDragUpdate` — A pointer in contact with the screen has moved further in the vertical direction.
  - `onVerticalDragEnd` — A pointer that was previously in contact with the screen and moving vertically is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.
- Horizontal dragging
  - `onHorizontalDragStart` — A pointer has contacted the screen and might begin to move horizontally.
  - `onHorizontalDragUpdate` — A pointer in contact with the screen has moved further in the horizontal direction.
  - `onHorizontalDragEnd` — A pointer that was previously in contact with the screen and moving horizontally is no longer in contact with the screen.

The following example shows a `GestureDetector` that rotates the Flutter logo on a double tap:

```
AnimationController controller;
CurvedAnimation curve;
```

```
@override
void initState() {
  controller = AnimationController(duration: const
Duration(milliseconds: 2000), vsync: this);
  curve = CurvedAnimation(parent: controller, curve:
Curves.easeIn);
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: GestureDetector(
          child: RotationTransition(
            turns: curve,
            child: FlutterLogo(
              size: 200.0,
            )),
          onDoubleTap: () {
            if (controller.isCompleted) {
              controller.reverse();
            } else {
              controller.forward();
            }
          },
        ),
      ),
    );
  }
}
```

Theming and text

How do I theme an app?

Out of the box, Flutter comes with a beautiful implementation of Material Design, which takes care of a lot of styling and theming needs that you would typically do.

To take full advantage of Material Components in your app, declare a top-level widget, MaterialApp, as the entry point to your application. MaterialApp is a convenience widget that wraps a number of widgets that are commonly required for applications implementing Material Design. It builds upon a WidgetsApp by adding Material specific functionality.

But Flutter is flexible and expressive enough to implement any design language. On iOS, you can use the Cupertino libraryto produce an interface that adheres to the Human Interface Guidelines. For the full set of these widgets, see theCupertino widgets gallery.

You can also use a `WidgetApp` as your app widget, which provides some of the same functionality, but is not as rich as `MaterialApp`.

To customize the colors and styles of any child components, pass a `ThemeData` object to the `MaterialApp` widget. For example, in the code below, the primary swatch is set to blue and text selection color is red.

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textSelectionColor: Colors.red
      ),
      home: SampleAppPage(),
    );
  }
}
```

How do I set custom fonts on my Text widgets?

In iOS, you import any `ttf` font files into your project and create a reference in the `info.plist` file. In Flutter, place the font file in a folder and reference it in the `pubspec.yaml` file, similar to how you import images.

```
fonts:
   - family: MyCustomFont
     fonts:
       - asset: fonts/MyCustomFont.ttf
       - style: italic
```

Then assign the font to your `Text` widget:

```
@override
```

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Sample App"),
    ),
    body: Center(
      child: Text(
        'This is a custom font text',
        style: TextStyle(fontFamily: 'MyCustomFont'),
      ),
    ),
  );
}
```

How do I style my Text widgets?

Along with fonts, you can customize other styling elements on a `Text` widget. The style parameter of a `Text` widget takes a `TextStyle` object, where you can customize many parameters, such as:

- `color`
- `decoration`
- `decorationColor`
- `decorationStyle`
- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `hashCode`
- `height`
- `inherit`
- `letterSpacing`
- `textBaseline`
- `wordSpacing`

Form input

How do forms work in Flutter? How do I retrieve user input?

Given how Flutter uses immutable widgets with a separate state, you might be wondering how user input fits into the picture. On iOS, you usually query the widgets for their current values when it’s time to submit the user input, or action on it. How does that work in Flutter?

In practice forms are handled, like everything in Flutter, by specialized widgets. If you have a `TextField` or a `TextFormField`, you can supply a `TextEditingController` to retrieve user input:

```
class _MyFormState extends State<MyForm> {
  // Create a text controller and use it to retrieve the
current value.
  // of the TextField!
  final myController = TextEditingController();

  @override
  void dispose() {
    // Clean up the controller when disposing of the
Widget.
    myController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Retrieve Text Input'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: TextField(
          controller: myController,
        ),
      ),
      floatingActionButton: FloatingActionButton(
        // When the user presses the button, show an alert
dialog with the
        // text the user has typed into our text field.
        onPressed: () {
          return showDialog(
            context: context,
            builder: (context) {
              return AlertDialog(
                // Retrieve the text the user has typed in
using our
                // TextEditingController
                content: Text(myController.text),
              );
            },
          );
        },
        tooltip: 'Show me the value!',
        child: Icon(Icons.text_fields),
      ),
    );
  }
}
```

You can find more information and the full code listing in Récupérer la valeur d'un champ texte , from the Flutter Cookbook.

What is the equivalent of a placeholder in a text field?

In Flutter you can easily show a “hint” or a placeholder text for your field by adding an `InputDecoration` object to the decoration constructor parameter for the `Text` widget:

```
body: Center(
  child: TextField(
    decoration: InputDecoration(hintText: "This is a
hint"),
  ),
)
```

How do I show validation errors?

Just as you would with a “hint”, pass an `InputDecoration` object to the decoration constructor for the `Text` widget.

However, you don’t want to start off by showing an error. Instead, when the user has entered invalid data, update the state, and pass a new `InputDecoration` object.

```
class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Sample App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() =>
_SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  String _errorText;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Sample App"),
      ),
      body: Center(
        child: TextField(
          onSubmitted: (String text) {
            setState(() {
              if (!isEmail(text)) {
                _errorText = 'Error: This is not an
email';
              } else {
                _errorText = null;
              }
            });
          },
          decoration: InputDecoration(hintText: "This is a
hint", errorText: _getErrorText()),
        ),
      ),
    );
  }

  _getErrorText() {
    return _errorText;
  }

  bool isEmail(String emailString) {
    String emailRegexp =

r'^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\
".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-
9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

    RegExp regExp = RegExp(emailRegexp);

    return regExp.hasMatch(emailString);
  }
}
```

Interacting with hardware, third party services and the platform

How do I interact with the platform, and with platform native code?

Flutter doesn't run code directly on the underlying platform; rather, the Dart code that makes up a Flutter app is run natively on the device, "sidestepping" the SDK provided by the platform. That means, for example, when you perform a network request in Dart, it runs directly in the Dart context. You don't use the Android or iOS APIs you normally take advantage of when writing native apps. Your Flutter app is still hosted in a native app's `ViewController` as a view, but you don't have direct access to the `ViewController` itself, or the native framework.

This doesn't mean Flutter apps cannot interact with those native APIs, or with any native code you have. Flutter provides platform channels, that communicate and exchange data with the `ViewController` that hosts your Flutter view. Platform channels are essentially an asynchronous messaging mechanism that bridge the Dart code with the host `ViewController`and the iOS framework it runs on. You can use platform channels to execute a method on the native side, or to retrieve some data from the device's sensors, for example.

In addition to directly using platform channels, you can use a variety of pre-made plugins that encapsulate the native and Dart code for a specific goal. For example, you can use a plugin to access the camera roll and the device camera directly from Flutter, without having to write your own integration. Plugins are found on the Pub site, Dart and Flutter's open source package repository. Some packages might support native integrations on iOS, or Android, or both.

If you can’t find a plugin on Pub that fits your needs, you can write your own and publish it on Pub.

How do I access the GPS sensor?

Use the `geolocator` community plugin.

How do I access the camera?

The `image_picker` plugin is popular for accessing the camera.

How do I log in with Facebook?

To log in with Facebook, use the `flutter_facebook_login` community plugin.

How do I use Firebase features?

Most Firebase functions are covered by first party plugins. These plugins are first-party integrations, maintained by the Flutter team:

- `firebase_admob` for Firebase AdMob
- `firebase_analytics` for Firebase Analytics
- `firebase_auth` for Firebase Auth
- `firebase_core` for Firebase’s Core package
- `firebase_database` for Firebase RTDB
- `firebase_storage` for Firebase Cloud Storage
- `firebase_messaging` for Firebase Messaging (FCM)
- `cloud_firestore` for Firebase Cloud Firestore

You can also find some third-party Firebase plugins on the Pub site that cover areas not directly covered by the first-party plugins.

How do I build my own custom native integrations?

If there is platform-specific functionality that Flutter or its community Plugins are missing, you can build your own following the developing packages and plugins page.

Flutter’s plugin architecture, in a nutshell, is much like using an Event bus in Android: you fire off a message and let the receiver process and emit a result back to you. In this case, the receiver is code running on the native side on Android or iOS.

Databases and local storage

How do I access UserDefault in Flutter?

In iOS, you can store a collection of key-value pairs using a property list, known as the `UserDefaults`.

In Flutter, access equivalent functionality using the Shared Preferences plugin. This plugin wraps the functionality of both `UserDefaults` and the Android equivalent, `SharedPreferences`.

What is the equivalent to CoreData in Flutter?

In iOS, you can use CoreData to store structured data. This is simply a layer on top of an SQL database, making it easier to make queries that relate to your models.

In Flutter, access this functionality using the SQFlite plugin.

Debugging

What tools can I use to debug my app in Flutter?

Use the DevTools suite for debugging Flutter or Dart apps.

DevTools includes support for profiling, examining the heap, inspecting the widget tree, logging diagnostics, debugging, observing executed lines of code, debugging memory leaks and memory fragmentation. For more information, see theDevTools documentation.

Notifications

How do I set up push notifications?

In iOS, you need to register your app on the developer portal to allow push notifications.

In Flutter, access this functionality using the `firebase_messaging` plugin.

For more information on using the Firebase Cloud Messaging API, see the `firebase_messaging` plugin documentation.

Flutter for web developers

This page is for users who are familiar with the HTML and CSS syntax for arranging components of an application's UI. It maps HTML/CSS code snippets to their Flutter/Dart code equivalents.

The examples assume:

- The HTML document starts with `<!DOCTYPE html>`, and the CSS box model for all HTML elements is set to `border-box`, for consistency with the Flutter model.

```
{
  box-sizing: border-box;
}
```

- In Flutter, the default styling of the "Lorem ipsum" text is defined by the `bold24Roboto` variable as follows, to keep the syntax simple:

```
TextStyle bold24Roboto = TextStyle(
  color: Colors.white,
  fontSize: 24.0,
  fontWeight: FontWeight.w900,
);
```

How is react-style, or *declarative*, programming different than the traditional imperative style? For a comparison, see Introduction to declarative UI.

Performing basic layout operations

The following examples show how to perform the most common UI layout tasks.

Styling and aligning text

Font style, size, and other text attributes that CSS handles with the font and color properties are individual properties of aTextStyle child of a Text widget.

In both HTML and Flutter, child elements or widgets are anchored at the top left, by default.

```
<div class="greybox">
    Lorem ipsum
</div>
```

```
.greybox {
      background-color: #e0e0e0; /* grey 300 */
      width: 320px;
      height: 240px;
      font: 900 24px Georgia;
    }
var container = Container( // grey box
  child: Text(
    "Lorem ipsum",
    style: TextStyle(
      fontSize: 24.0,
      fontWeight: FontWeight.w900,
      fontFamily: "Georgia",
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Setting background color

In Flutter, you set the background color using a Container's `decoration` property.

The CSS examples use the hex color equivalents to the Material color palette.

```
<div class="greybox">
  Lorem ipsum
</div>

.greybox {
      background-color: #e0e0e0;  /* grey 300 */
      width: 320px;
      height: 240px;
      font: 900 24px Roboto;
    }
var container = Container( // grey box
  child: Text(
    "Lorem ipsum",
    style: bold24Roboto,
  ),
  width: 320.0,
  height: 240.0,
  decoration: BoxDecoration(
    color: Colors.grey[300],
  ),
);
```

Centering components

A Center widget centers its child both horizontally and vertically.

To accomplish a similar effect in CSS, the parent element uses either a flex or table-cell display behavior. The examples on this page show the flex behavior.

```
<div class="greybox">
  Lorem ipsum
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
var container = Container( // grey box
  child: Center(
    child: Text(
      "Lorem ipsum",
      style: bold24Roboto,
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Setting container width

To specify the width of a Container widget, use its `width` property. This is a fixed width, unlike the CSS max-width property that adjusts the container width up to a maximum value. To mimic that effect in Flutter, use the `constraints` property of the Container. Create a new BoxConstraints widget with a `minWidth` or `maxWidth`.

For nested Containers, if the parent's width is less than the child's width, the child Container sizes itself to match the parent.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  width: 100%;
  max-width: 240px;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
      ),
      padding: EdgeInsets.all(16.0),
      width: 240.0, //max-width is 240.0
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Manipulating position and size

The following examples show how to perform more complex operations on widget position, size, and background.

Setting absolute position

By default, widgets are positioned relative to their parent.

To specify an absolute position for a widget as x-y coordinates, nest it in a Positioned widget that is, in turn, nested in aStack widget.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  position: relative;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  position: absolute;
  top: 24px;
  left: 24px;
}
var container = Container( // grey box
  child: Stack(
    children: [
      Positioned( // red box
        child: Container(
          child: Text(
            "Lorem ipsum",
            style: bold24Roboto,
          ),
          decoration: BoxDecoration(
            color: Colors.red[400],
          ),
          padding: EdgeInsets.all(16.0),
        ),
        left: 24.0,
        top: 24.0,
      ),
    ],
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Rotating components

To rotate a widget, nest it in a Transform widget. Use the Transform widget's `alignment` and `origin` properties to specify the transform origin (fulcrum) in relative and absolute terms, respectively.

For a simple 2D rotation, the widget is rotated on the Z axis using radians. (degrees × π / 180)

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  transform: rotate(15deg);
}
var container = Container( // gray box
  child: Center(
    child:  Transform(
      child:  Container( // red box
        child: Text(
          "Lorem ipsum",
          style: bold24Roboto,
          textAlign: TextAlign.center,
        ),
        decoration: BoxDecoration(
          color: Colors.red[400],
        ),
        padding: EdgeInsets.all(16.0),
      ),
      alignment: Alignment.center,
      transform: Matrix4.identity()
        ..rotateZ(15 * 3.1415927 / 180),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Scaling components

To scale a widget up or down, nest it in a Transform widget. Use the Transform widget's `alignment` and `origin` properties to specify the transform origin (fulcrum) in relative or absolute terms, respectively.

For a simple scaling operation along the x-axis, create a new Matrix4 identity object and use its scale() method to specify the scaling factor.

When you scale a parent widget, its child widgets are scaled accordingly.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  transform: scale(1.5);
}
```

```
var container = Container( // gray box
  child: Center(
    child:  Transform(
      child:  Container( // red box
        child: Text(
          "Lorem ipsum",
          style: bold24Roboto,
          textAlign: TextAlign.center,
        ),
        decoration: BoxDecoration(
          color: Colors.red[400],
        ),
        padding: EdgeInsets.all(16.0),
      ),
      alignment: Alignment.center,
      transform: Matrix4.identity()
        ..scale(1.5),
     ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Applying a linear gradient

To apply a linear gradient to a widget's background, nest it in a Container widget. Then use the Container widget's `decoration` property to create a BoxDecoration object, and use BoxDecoration's `gradient` property to transform the background fill.

The gradient "angle" is based on the Alignment (x, y) values:

- If the beginning and ending x values are equal, the gradient is vertical (0° | 180°).
- If the beginning and ending y values are equal, the gradient is horizontal (90° | 270°).

Vertical Gradient

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  padding: 16px;
  color: #ffffff;
  background: linear-gradient(180deg, #ef5350, rgba(0, 0,
0, 0) 80%);
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
      ),
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: const Alignment(0.0, -1.0),
          end: const Alignment(0.0, 0.6),
          colors: <Color>[
            const Color(0xffef5350),
            const Color(0x00ef5350)
          ],
        ),
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Horizontal gradient

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>
```

```
.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  padding: 16px;
  color: #ffffff;
  background: linear-gradient(90deg, #ef5350, rgba(0, 0,
0, 0) 80%);
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
      ),
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: const Alignment(-1.0, 0.0),
          end: const Alignment(0.6, 0.0),
          colors: <Color>[
            const Color(0xffef5350),
            const Color(0x00ef5350)
          ],
        ),
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Manipulating shapes

The following examples show how to make and customize shapes.

Rounding corners

To round the corners of a rectangular shape, use the `borderRadius` property of a BoxDecoration object. Create a newBorderRadius object that specifies the radii for rounding each corner.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* gray 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  border-radius: 8px;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red circle
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
        borderRadius: BorderRadius.all(
          const Radius.circular(8.0),
        ),
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Adding box shadows

In CSS you can specify shadow offset and blur in shorthand, using the box-shadow property. This example shows two box shadows, with properties:

- `xOffset: 0px, yOffset: 2px, blur: 4px, color: black @80% alpha`
- `xOffset: 0px, yOffset: 06x, blur: 20px, color: black @50% alpha`

In Flutter, each property and value is specified separately. Use the `boxShadow` property of BoxDecoration to create a list ofBoxShadow widgets. You can define one or multiple BoxShadow widgets, which can be stacked to customize the shadow depth, color, etc.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.8),
              0 6px 20px rgba(0, 0, 0, 0.5);
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
        boxShadow: <BoxShadow>[
          BoxShadow (
            color: const Color(0xcc000000),
            offset: Offset(0.0, 2.0),
            blurRadius: 4.0,
          ),
          BoxShadow (
            color: const Color(0x80000000),
            offset: Offset(0.0, 6.0),
            blurRadius: 20.0,
          ),
        ],
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  decoration: BoxDecoration(
    color: Colors.grey[300],
  ),
  margin: EdgeInsets.only(bottom: 16.0),
);
```

Making circles and ellipses

Making a circle in CSS requires a workaround of applying a border-radius of 50% to all four sides of a rectangle, though there are basic shapes.

While this approach is supported with the `borderRadius` property of BoxDecoration, Flutter provides a `shape` property with BoxShape enum for this purpose.

```
<div class="greybox">
  <div class="redcircle">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* gray 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redcircle {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  text-align: center;
  width: 160px;
  height: 160px;
  border-radius: 50%;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red circle
      child: Text(
        "Lorem ipsum",
        style: bold24Roboto,
        textAlign: TextAlign.center,
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
        shape: BoxShape.circle,
      ),
      padding: EdgeInsets.all(16.0),
      width: 160.0,
      height: 160.0,
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Manipulating text

The following examples show how to specify fonts and other text attributes. They also show how to transform text strings, customize spacing, and create excerpts.

Adjusting text spacing

In CSS you specify the amount of white space between each letter or word by giving a length value for the letter-spacing and word-spacing properties, respectively. The amount of space can be in px, pt, cm, em, etc.

In Flutter, you specify white space as logical pixels (negative values are allowed) for the `letterSpacing` and `wordSpacing`properties of a TextStyle child of a Text widget.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  letter-spacing: 4px;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum",
        style: TextStyle(
          color: Colors.white,
          fontSize: 24.0,
          fontWeight: FontWeight.w900,
          letterSpacing: 4.0,
        ),
      ),
      decoration: BoxDecoration(
        color: Colors.red[400],
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Making inline formatting changes

A Text widget lets you display text with the same formatting characteristics. To display text that uses multiple styles (in this example, a single word with emphasis), use a RichText widget instead. Its `text` property can specify one or more TextSpanwidgets that can be individually styled.

In the following example, “Lorem” is in a TextSpan widget with the default (inherited) text styling, and “ipsum” is in a separate TextSpan with custom styling.

```
<div class="greybox">
  <div class="redbox">
    Lorem <em>ipsum</em>
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
}
 .redbox em {
  font: 300 48px Roboto;
  font-style: italic;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child:  RichText(
        text: TextSpan(
          style: bold24Roboto,
          children: <TextSpan>[
            TextSpan(text: "Lorem "),
            TextSpan(
              text: "ipsum",
              style: TextStyle(
                fontWeight: FontWeight.w300,
                fontStyle: FontStyle.italic,
                fontSize: 48.0,
              ),
            ),
          ],
        ),
      ),
      decoration: BoxDecoration(
        backgroundColor: Colors.red[400],
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Creating text excerpts

An excerpt displays the initial line(s) of text in a paragraph, and handles the overflow text, often using an ellipsis. In HTML/CSS an excerpt can be no longer than one line. Truncating after multiple lines requires some JavaScript code.

In Flutter, use the `maxLines` property of a Text widget to specify the number of lines to include in the excerpt, and the`overflow` property for handling overflow text.

```
<div class="greybox">
  <div class="redbox">
    Lorem ipsum dolor sit amet, consec etur
  </div>
</div>

.greybox {
  background-color: #e0e0e0; /* grey 300 */
  width: 320px;
  height: 240px;
  font: 900 24px Roboto;
  display: flex;
  align-items: center;
  justify-content: center;
}
.redbox {
  background-color: #ef5350; /* red 400 */
  padding: 16px;
  color: #ffffff;
  overflow: hidden;
  text-overflow: ellipsis;
  white-space: nowrap;
}
var container = Container( // grey box
  child: Center(
    child: Container( // red box
      child: Text(
        "Lorem ipsum dolor sit amet, consec etur",
        style: bold24Roboto,
        overflow: TextOverflow.ellipsis,
        maxLines: 1,
      ),
      decoration: BoxDecoration(
        backgroundColor: Colors.red[400],
      ),
      padding: EdgeInsets.all(16.0),
    ),
  ),
  width: 320.0,
  height: 240.0,
  color: Colors.grey[300],
);
```

Layouts in Flutter

1. Documentation
2. Développement
3. Interface utilisateur
4. Layout

À quoi ça sert?

- Les widgets sont des classes utilisées pour construire des interfaces utilisateur.
- Les widgets sont utilisés pour les éléments de présentation et d'interface utilisateur.
- Composez des widgets simples pour construire des widgets complexes.

Le mécanisme de présentation de Flutter est principalement basé sur les widgets. Dans Flutter, presque tout est un widget - même les modèles de présentation sont des widgets. Les images, les icônes et le texte que vous voyez dans une application Flutter sont tous des widgets. Mais les choses que vous ne voyez pas sont aussi des widgets, tels que les lignes, les colonnes, et des grilles qui organisent, contraignent et alignent les widgets visibles.

Vous créez une mise en page en composant des widgets pour construire des widgets plus complexes. Par exemple, la première capture d'écran ci-dessous montre 3 icônes avec une étiquette sous chacune d'elles:

La deuxième capture d'écran affiche la disposition visuelle, montrant une rangée de 3 colonnes où chaque colonne contient une icône et une étiquette.

Remarque: La plupart des captures d'écran de ce tutoriel sont affichées avec `debugPaintSizeEnabled`positionné sur "true" pour que vous puissiez voir la disposition visuelle. Pour plus d'informations, voir Debugging layout issues visually, une section de Using the Flutter inspector.

Voici un diagramme de l'arborescence des widgets pour cette interface utilisateur:

Most of this should look as you might expect, but you might be wondering about the containers (shown in pink). Containeris a widget class that allows you to customize its child widget. Use a `Container` when you want to add padding, margins, borders, or background color, to name some of its capabilities.

In this example, each Text widget is placed in a `Container` to add margins. The entire Row is also placed in a `Container` to add padding around the row.

The rest of the UI in this example is controlled by properties. Set an Icon's color using its `color` property. Use the `Text.style` property to set the font, its color, weight, and so on. Columns and rows have properties that allow you to specify how their children are aligned vertically or horizontally, and how much space the children should occupy.

Disposer un widget

How do you layout a single widget in Flutter? This section shows you how to create and display a simple widget. It also shows the entire code for a simple Hello World app.

In Flutter, it takes only a few steps to put text, an icon, or an image on the screen.

1. Select a layout widget

Choose from a variety of widgets de mise en page based on how you want to align or constrain the visible widget, as these characteristics are typically passed on to the contained widget.

This example uses Center which centers its content horizontally and vertically.

2. Create a visible widget

For example, create a Text widget:

```
Text('Hello World'),
```

Créé un dossier Image widget:

```
Image.asset(
  'images/lake.jpg',
  fit: BoxFit.cover,
),
```

Create an Icon widget:

```
Icon(
  Icons.star,
  color: Colors.red[500],
),
```

3. Add the visible widget to the layout widget

All layout widgets have either of the following:

- A `child` property if they take a single child – for example, `Center` ou `Container`
- A `children` property if they take a list of widgets – for example, `Row`, `Column`, `ListView`, or `Stack`.

Add the `Text` widget to the `Center` widget:

```
Center(
  child: Text('Hello World'),
),
```

4. Add the layout widget to the page

A Flutter app is itself a widget, and most widgets have a build() method. Instantiating and returning a widget in the app's `build()` method displays the widget.

Material apps

For a `Material` app, you can use a Scaffold widget; it provides a default banner, background color, and has API for adding drawers, snack bars, and bottom sheets. Then you can add the `Center` widget directly to the `body` property for the home page.

lib/main.dart (MyApp)

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

Note: The Material library implements widgets that follow Material Design principles. When designing your UI, you can exclusively use widgets from the standard widgets library, or you can use widgets from the Material library. You can mix widgets from both libraries, you can customize existing widgets, or you can build your own set of custom widgets.

Non-Material apps

For a non-Material app, you can add the `Center` widget to the app's `build()`:

lib/main.dart (MyApp)

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(color: Colors.white),
      child: Center(
        child: Text(
          'Hello World',
          textDirection: TextDirection.ltr,
          style: TextStyle(
            fontSize: 32,
            color: Colors.black87,
          ),
        ),
      ),
    );
  }
}
```

By default a non-Material app doesn't include an `AppBar`, title, or background color. If you want these features in a non-Material app, you have to build them yourself. This app changes the background color to white and the text to dark grey to mimic a Material app.

That's it! When you run the app, you should see *Hello World*.

App source code:

- Material app
- Non-Material app

Lay out multiple widgets vertically and horizontally

One of the most common layout patterns is to arrange widgets vertically or horizontally. You can use a Row widget to arrange widgets horizontally, and a Column widget to arrange widgets vertically.

À quoi ça sert?

- Row and Column are two of the most commonly used layout patterns.
- Row and Column each take a list of child widgets.
- A child widget can itself be a Row, Column, or other complex widget.
- You can specify how a Row or Column aligns its children, both vertically and horizontally.
- You can stretch or constrain specific child widgets.
- You can specify how child widgets use the Row's or Column's available space.

To create a row or column in Flutter, you add a list of children widgets to a Row or Column widget. In turn, each child can itself be a row or column, and so on. The following example shows how it is possible to nest rows or columns inside of rows or columns.

This layout is organized as a Row. The row contains two children: a column on the left, and an image on the right:

The left column's widget tree nests rows and columns.

You'll implement some of Pavlova's layout code in Nesting rows and columns.

Note: Row and Column are basic primitive widgets for horizontal and vertical layouts—these low-level widgets allow for maximum customization. Flutter also offers specialized, higher level widgets that might be sufficient for your needs. For example, instead of Row you might prefer ListTile, an easy-to-use widget with properties for leading and trailing icons, and up to 3 lines of text. Instead of Column, you might prefer ListView, a column-like layout that automatically scrolls if its content is too long to fit the available space. For more information, see Common layout widgets.

Aligning widgets

You control how a row or column aligns its children using the `mainAxisAlignment` and `crossAxisAlignment` properties. For a row, the main axis runs horizontally and the cross axis runs vertically. For a column, the main axis runs vertically and the cross axis runs horizontally.

The MainAxisAlignment and CrossAxisAlignment classes offer a variety of constants for controlling alignment.

Note: When you add images to your project, you need to update the pubspec file to access them—this example uses `Image.asset` to display the images. For more information, see this example's pubspec.yaml file, or Adding Assets and Images in Flutter. You don't need to do this if you're referencing online images using `Image.network`.

In the following example, each of the 3 images is 100 pixels wide. The render box (in this case, the entire screen) is more than 300 pixels wide, so setting the main axis alignment to `spaceEvenly` divides the free horizontal space evenly between, before, and after each image.

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Image.asset('images/pic1.jpg'),
    Image.asset('images/pic2.jpg'),
    Image.asset('images/pic3.jpg'),
  ],
);
```

App source: row_column

Columns work the same way as rows. The following example shows a column of 3 images, each is 100 pixels high. The height of the render box (in this case, the entire screen) is more than 300 pixels, so setting the main axis alignment to `spaceEvenly` divides the free vertical space evenly between, above, and below each image.

```
Column(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Image.asset('images/pic1.jpg'),
    Image.asset('images/pic2.jpg'),
    Image.asset('images/pic3.jpg'),
  ],
);
```

App source: row_column

Sizing widgets

When a layout is too large to fit a device, a yellow and black striped pattern appears along the affected edge. Here is an example of a row that is too wide:

Widgets can be sized to fit within a row or column by using the Expanded widget. To fix the previous example where the row of images is too wide for its render box, wrap each image with an `Expanded` widget.

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

App source: sizing

Perhaps you want a widget to occupy twice as much space as its siblings. For this, use the `Expanded` widget `flex`property, an integer that determines the flex factor for a widget. The default flex factor is 1. The following code sets the flex factor of the middle image to 2:

```
Row(
  crossAxisAlignment: CrossAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      flex: 2,
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);
```

App source: sizing

Packing widgets

By default, a row or column occupies as much space along its main axis as possible, but if you want to pack the children closely together, set its `mainAxisSize` to `MainAxisSize.min`. The following example uses this property to pack the star icons together.

```
Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.black),
    Icon(Icons.star, color: Colors.black),
  ],
)
```

App source: pavlova

Nesting rows and columns

The layout framework allows you to nest rows and columns inside of rows and columns as deeply as you need. Let’s look the code for the outlined section of the following layout:

The outlined section is implemented as two rows. The ratings row contains five stars and the number of reviews. The icons row contains three columns of icons and text.

The widget tree for the ratings row:

The `ratings` variable creates a row containing a smaller row of 5 star icons, and text:

```
var stars = Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.black),
    Icon(Icons.star, color: Colors.black),
  ],
);

final ratings = Container(
  padding: EdgeInsets.all(20),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      stars,
      Text(
        '170 Reviews',
        style: TextStyle(
          color: Colors.black,
          fontWeight: FontWeight.w800,
          fontFamily: 'Roboto',
          letterSpacing: 0.5,
          fontSize: 20,
        ),
      ),
    ],
  ),
);
```

Actuce: To minimize the visual confusion that can result from heavily nested layout code, implement pieces of the UI in variables and functions.

The icons row, below the ratings row, contains 3 columns; each column contains an icon and two lines of text, as you can see in its widget tree:

The `iconList` variable defines the icons row:

```
final descTextStyle = TextStyle(
  color: Colors.black,
  fontWeight: FontWeight.w800,
  fontFamily: 'Roboto',
  letterSpacing: 0.5,
  fontSize: 18,
  height: 2,
);

// DefaultTextStyle.merge() allows you to create a default
text
// style that is inherited by its child and all subsequent
children.
final iconList = DefaultTextStyle.merge(
  style: descTextStyle,
  child: Container(
    padding: EdgeInsets.all(20),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        Column(
          children: [
            Icon(Icons.kitchen, color: Colors.green[500]),
            Text('PREP:'),
            Text('25 min'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.timer, color: Colors.green[500]),
            Text('COOK:'),
            Text('1 hr'),
          ],
        ),
        Column(
          children: [
            Icon(Icons.restaurant, color:
Colors.green[500]),
            Text('FEEDS:'),
            Text('4-6'),
          ],
        ),
      ],
    ),
  ),
);
```

The `leftColumn` variable contains the ratings and icons rows, as well as the title and text that describes the Pavlova:

```
final leftColumn = Container(
  padding: EdgeInsets.fromLTRB(20, 30, 20, 20),
  child: Column(
    children: [
      titleText,
      subTitle,
      ratings,
      iconList,
    ],
  ),
);
```

The left column is placed in a `Container` to constrain its width. Finally, the UI is constructed with the entire row (containing the left column and the image) inside a `Card`.

The Pavlova image is from Pixabay. You can embed an image from the net using `Image.network()` but, for this example, the image is saved to an images directory in the project, added to the fichier pubspec and accessed using `Images.asset()`. For more information, see Adding assets and images.

```
body: Center(
  child: Container(
    margin: EdgeInsets.fromLTRB(0, 40, 0, 30),
    height: 600,
    child: Card(
      child: Row(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Container(
            width: 440,
            child: leftColumn,
          ),
          mainImage,
        ],
      ),
    ),
  ),
),
```

Tip: The Pavlova example runs best horizontally on a wide device, such as a tablet. If you are running this example in the iOS simulator, you can select a different device using the **Hardware** > **Device** menu. For this example, we recommend the iPad Pro. You can change its orientation to landscape mode using **Hardware** > **Rotate**. You can also change the size of the simulator window (without changing the number of logical pixels) using **Window** > **Scale**.

App source: pavlova

Common layout widgets

Flutter has a rich library of layout widgets. Here are a few of those most commonly used. The intent is to get you up and running as quickly as possible, rather than overwhelm you with a complete list. For information on other available widgets, refer to the Widget catalog, or use the Search box in the API reference docs. Also, the widget pages in the API docs often make suggestions about similar widgets that might better suit your needs.

The following widgets fall into two categories: standard widgets from the widgets library, and specialized widgets from the Material library. Any app can use the widgets library but only Material apps can use the Material Components library.

Standard widgets

- Container: Adds padding, margins, borders, background color, or other decorations to a widget.
- GridView: Lays widgets out as a scrollable grid.
- ListView: Lays widgets out as a scrollable list.
- Stack: Overlaps a widget on top of another.

Material widgets

- Card: Organizes related info into a box with rounded corners and a drop shadow.
- ListTile: Organizes up to 3 lines of text, and optional leading and trailing icons, into a row.

Container

Many layouts make liberal use of Containers to separate widgets using padding, or to add borders or margins. You can change the device's background by placing the entire layout into a `Container` and changing its background color or image.

Summary (Container)

- Add padding, margins, borders
- Change background color or image
- Contains a single child widget, but that child can be a Row, Column, or even the root of a widget tree

Examples (Container)

This layout consists of a column with two rows, each containing 2 images. A Container is used to change the background color of the column to a lighter grey.

```
Widget _buildImageColumn() => Container(
      decoration: BoxDecoration(
        color: Colors.black26,
      ),
      child: Column(
        children: [
          _buildImageRow(1),
          _buildImageRow(3),
        ],
      ),
    );
```

A `Container` is also used to add a rounded border and margins to each image:

```
Widget _buildDecoratedImage(int imageIndex) => Expanded(
      child: Container(
        decoration: BoxDecoration(
          border: Border.all(width: 10, color:
Colors.black38),
          borderRadius: const BorderRadius.all(const
Radius.circular(8)),
        ),
        margin: const EdgeInsets.all(4),
        child: Image.asset('images/pic$imageIndex.jpg'),
      ),
    );

Widget _buildImageRow(int imageIndex) => Row(
      children: [
        _buildDecoratedImage(imageIndex),
        _buildDecoratedImage(imageIndex + 1),
      ],
    );
```

You can find more `Container` examples in the tutorial and the Flutter Gallery.

App source: container

GridView

Use GridView to lay widgets out as a two-dimensional list. `GridView` provides two pre-fabricated lists, or you can build your own custom grid. When a `GridView` detects that its contents are too long to fit the render box, it automatically scrolls.

Summary (GridView)

- Lays widgets out in a grid
- Detects when the column content exceeds the render box and automatically provides scrolling
- Build your own custom grid, or use one of the provided grids:
    - `GridView.count` allows you to specify the number of columns
    - `GridView.extent` allows you to specify the maximum pixel width of a tile

Note: When displaying a two-dimensional list where it's important which row and column a cell occupies (for example, it's the entry in the "calorie" column for the "avocado" row), use Table or DataTable.

Examples (GridView)

Uses `GridView.extent` to create a grid with tiles a maximum 150 pixels wide.

App source: grid_and_list

Uses `GridView.count` to create a grid that's 2 tiles wide in portrait mode, and 3 tiles wide in landscape mode. The titles are created by setting the `footer` property for each GridTile.

Code Dart grid_list_demo.dart from the Flutter Gallery

```
Widget _buildGrid() => GridView.extent(
    maxCrossAxisExtent: 150,
    padding: const EdgeInsets.all(4),
    mainAxisSpacing: 4,
    crossAxisSpacing: 4,
    children: _buildGridTileList(30));

// The images are saved with names pic0.jpg,
pic1.jpg...pic29.jpg.
// The List.generate() constructor allows an easy way to
create
// a list when objects have a predictable naming pattern.
List<Container> _buildGridTileList(int count) =>
List.generate(
    count, (i) => Container(child:
Image.asset('images/pic$i.jpg')));
```

ListView

ListView, a column-like widget, automatically provides scrolling when its content is too long for its render box.

Summary (ListView)

- A specialized Column for organizing a list of boxes
- Can be laid out horizontally or vertically
- Detects when its content won't fit and provides scrolling
- Less configurable than `Column`, but easier to use and supports scrolling

Examples (ListView)

Uses `ListView` to display a list of businesses using `ListTile`s.
A `Divider` separates the theaters from the restaurants.

App source: grid_and_list

Uses `ListView` to display the Colors from the Material Design palette for a particular color family.

Dart code: colors_demo.dart from the Flutter Gallery

```
Widget _buildList() => ListView(
      children: [
        _tile('CineArts at the Empire', '85 W Portal Ave',
Icons.theaters),
        _tile('The Castro Theater', '429 Castro St',
Icons.theaters),
        _tile('Alamo Drafthouse Cinema', '2550 Mission
St', Icons.theaters),
        _tile('Roxie Theater', '3117 16th St',
Icons.theaters),
        _tile('United Artists Stonestown Twin', '501
Buckingham Way',
            Icons.theaters),
        _tile('AMC Metreon 16', '135 4th St #3000',
Icons.theaters),
        Divider(),
        _tile('Kescaped_code#39;s Kitchen', '757 Monterey
Blvd', Icons.restaurant),
        _tile('Emmyescaped_code#39;s Restaurant', '1923
Ocean Ave', Icons.restaurant),
        _tile(
            'Chaiya Thai Restaurant', '272 Claremont
Blvd', Icons.restaurant),
        _tile('La Ciccia', '291 30th St',
Icons.restaurant),
      ],
    );

ListTile _tile(String title, String subtitle, IconData
icon) => ListTile(
      title: Text(title,
          style: TextStyle(
            fontWeight: FontWeight.w500,
            fontSize: 20,
          )),
      subtitle: Text(subtitle),
      leading: Icon(
        icon,
        color: Colors.blue[500],
      ),
    );
```

Stack

Use Stack to arrange widgets on top of a base widget—often an image. The widgets can completely or partially overlap the base widget.

Summary (Stack)

- Use for widgets that overlap another widget
- The first widget in the list of children is the base widget; subsequent children are overlaid on top of that base widget
- A `Stack`'s content can't scroll
- You can choose to clip children that exceed the render box

Examples (Stack)

Uses `Stack` to overlay a `Container` (that displays its `Text` on a translucent black background) on top of a `CircleAvatar`. The `Stack` offsets the text using the `alignment` property and `Alignment`s.

App source: card_and_stack

Uses `Stack` to overlay a gradient to the top of the image. The gradient ensures that the toolbar's icons are distinct against the image.

Dart code: contacts_demo.dart from the Flutter Gallery

```
Widget _buildStack() => Stack(
    alignment: const Alignment(0.6, 0.6),
    children: [
      CircleAvatar(
        backgroundImage: AssetImage('images/pic.jpg'),
        radius: 100,
      ),
      Container(
        decoration: BoxDecoration(
          color: Colors.black45,
        ),
        child: Text(
          'Mia B',
          style: TextStyle(
            fontSize: 20,
            fontWeight: FontWeight.bold,
            color: Colors.white,
          ),
        ),
      ),
    ],
  );
```

Card

A Card, from the Material library, contains related nuggets of information and can be composed from almost any widget, but is often used with ListTile. `Card` has a single child, but its child can be a column, row, list, grid, or other widget that supports multiple children. By default, a `Card` shrinks its size to 0 by 0 pixels. You can use SizedBox to constrain the size of a card.

In Flutter, a `Card` features slightly rounded corners and a drop shadow, giving it a 3D effect. Changing a `Card`'s `elevation`property allows you to control the drop shadow effect. Setting the elevation to 24, for example, visually lifts the `Card`further from the surface and causes the shadow to become more dispersed. For a list of supported elevation values, see Elevation in the Material guidelines. Specifying an unsupported value disables the drop shadow entirely.

Summary (Card)

- Implements a Material card
- Used for presenting related nuggets of information
- Accepts a single child, but that child can be a `Row`, `Column`, or other widget that holds a list of children
- Displayed with rounded corners and a drop shadow
- A `Card`'s content can't scroll
- From the Material library

Examples (Card)

A `Card` containing 3 ListTiles and sized by wrapping it with a `SizedBox`.
A `Divider` separates the first and second `ListTiles`.

App source: card_and_stack

A `Card` containing an image and text.

Dart code: cards_demo.dart from the Flutter Gallery

```
Widget _buildCard() => SizedBox(
    height: 210,
    child: Card(
      child: Column(
        children: [
          ListTile(
            title: Text('1625 Main Street',
                style: TextStyle(fontWeight:
FontWeight.w500)),
            subtitle: Text('My City, CA 99984'),
            leading: Icon(
              Icons.restaurant_menu,
              color: Colors.blue[500],
            ),
          ),
          Divider(),
          ListTile(
            title: Text('(408) 555-1212',
                style: TextStyle(fontWeight:
FontWeight.w500)),
            leading: Icon(
              Icons.contact_phone,
              color: Colors.blue[500],
            ),
          ),
          ListTile(
            title: Text('costa@example.com'),
            leading: Icon(
              Icons.contact_mail,
              color: Colors.blue[500],
            ),
          ),
        ],
      ),
    ),
  );
```

ListTile

Use ListTile, a specialized row widget from the Material library, for an easy way to create a row containing up to 3 lines of text and optional leading and trailing icons. `ListTile` is most commonly used in Card or ListView, but can be used elsewhere.

Summary (ListTile)

- A specialized row that contains up to 3 lines of text and optional icons
- Less configurable than `Row`, but easier to use
- From the Material library

Examples (ListTile)

A `Card` containing 3 `ListTiles`.

App source: card_and_stack

Uses `ListTile` to list 3 drop down button types.
Dart code: buttons_demo.dart from the Flutter Gallery

Videos

The following videos, part of the Flutter in Focus series, explain Stateless and Stateful widgets.

Flutter in Focus playlist

Each episode of the Widget of the Week series focuses on a widget. Several of them includes layout widgets.

Flutter Widget of the Week playlist

Other resources

The following resources might help when writing layout code.

- Layout tutorial
  Learn how to build a layout.
- Widget Overview
  Describes many of the widgets available in Flutter.
- HTML/CSS Analogs in Flutter
  For those familiar with web programming, this page maps HTML/CSS functionality to Flutter features.
- Flutter Gallery
  Demo app showcasing many Material Design widgets and other Flutter features.
- Flutter API documentation
  Reference documentation for all of the Flutter libraries.

- Dealing with Box Constraints in Flutter
  Discusses how widgets are constrained by their render boxes.
- Adding Assets and Images in Flutter
  Explains how to add images and other assets to your app's package.
- Zero to One with Flutter
  One person's experience writing his first Flutter app.

Adding interactivity to your Flutter app

1. Documentation
2. Développement
3. Interface utilisateur
4. Adding interactivity

Ce que vous allez apprendre

- How to respond to taps.
- How to create a custom widget.
- The difference between stateless and stateful widgets.

How do you modify your app to make it react to user input? In this tutorial, you'll add interactivity to an app that contains only non-interactive widgets. Specifically, you'll modify an icon to make it tappable by creating a custom stateful widget that manages two stateless widgets.

Layout tutorial showed you how to create the layout for the following screenshot.

The layout tutorial app

When the app first launches, the star is solid red, indicating that this lake has previously been favorited. The number next to the star indicates that 41 people have favorited this lake. After completing this tutorial, tapping the star removes its favorited status, replacing the solid star with an outline and decreasing the count. Tapping again favorites the lake, drawing a solid star and increasing the count.

To accomplish this, you'll create a single custom widget that includes both the star and the count, which are themselves widgets. Tapping the star changes state for both widgets, so the same widget should manage both.

You can get right to touching the code in Step 2: Subclass StatefulWidget. If you want to try different ways of managing state, skip to Managing state.

Stateful and stateless widgets

A widget is either stateful or stateless. If a widget can change—when a user interacts with it, for example—it's stateful.

A *stateless* widget never changes. Icon, IconButton, and Text are examples of stateless widgets. Stateless widgets subclass StatelessWidget.

A *stateful* widget is dynamic: for example, it can change its appearance in response to events triggered by user interactions or when it receives data. Checkbox, Radio, Slider, InkWell, Form, and TextField are examples of stateful widgets. Stateful widgets subclass StatefulWidget.

A widget's state is stored in a State object, separating the widget's state from its appearance. The state consists of values that can change, like a slider's current value or whether a checkbox is checked. When the widget's state changes, the state object calls `setState()`, telling the framework to redraw the widget.

Creating a stateful widget

What's the point?

- A stateful widget is implemented by two classes: a subclass of `StatefulWidget` and a subclass of `State`.
- The state class contains the widget's mutable state and the widget's `build()` method.
- When the widget's state changes, the state object calls `setState()`, telling the framework to redraw the widget.

In this section, you'll create a custom stateful widget. You'll replace two stateless widgets—the solid red star and the numeric count next to it—with a single custom stateful widget that manages a row with two children widgets: an `IconButton` and `Text`.

Implementing a custom stateful widget requires creating two classes:

- A subclass of `StatefulWidget` that defines the widget.
- A subclass of `State` that contains the state for that widget and defines the widget's `build()` method.

This section shows you how to build a stateful widget, called `FavoriteWidget`, for the lakes app. After setting up, your first step is choosing how state is managed for `FavoriteWidget`.

Step 0: Get ready

If you’ve already built the app in Layout tutorial (step 6), skip to the next section.

1. Make sure you’ve set up your environment.
2. Create a basic “Hello World” Flutter app.
3. Replace the `lib/main.dart` file with main.dart.
4. Replace the `pubspec.yaml` file with pubspec.yaml.
5. Create an `images` directory in your project, and add lake.jpg.

Once you have a connected and enabled device, or you’ve launched the iOS simulator (part of the Flutter install), you are good to go!

Step 1: Decide which object manages the widget’s state

A widget’s state can be managed in several ways, but in our example the widget itself, `FavoriteWidget`, will manage its own state. In this example, toggling the star is an isolated action that doesn’t affect the parent widget or the rest of the UI, so the widget can handle its state internally.

Learn more about the separation of widget and state, and how state might be managed, in Managing state.

Step 2: Subclass StatefulWidget

The `FavoriteWidget` class manages its own state, so it overrides `createState()` to create a `State` object. The framework calls `createState()` when it wants to build the widget. In this example, `createState()` returns an instance of `_FavoriteWidgetState`, which you’ll implement in the next step.

lib/main.dart (FavoriteWidget)

```
class FavoriteWidget extends StatefulWidget {
  @override
  _FavoriteWidgetState createState() =>
_FavoriteWidgetState();
}
```

Remarque: Members or classes that start with an underscore (_) are private. For more information, see Libraries and visibility, a section in the Dart language tour.

Step 3: Subclass State

The `_FavoriteWidgetState` class stores the mutable data that can change over the lifetime of the widget. When the app first launches, the UI displays a solid red star, indicating that the lake has “favorite” status, along with 41 likes. These values are stored in the `_isFavorited` and `_favoriteCount` fields:

lib/main.dart (_FavoriteWidgetState fields)

```
class _FavoriteWidgetState extends State<FavoriteWidget> {
  bool _isFavorited = true;
  int _favoriteCount = 41;
  // ···
}
```

The class also defines a `build()` method, which creates a row containing a red `IconButton`, and `Text`. You use IconButton(instead of `Icon`) because it has an `onPressed` property that defines the callback function (`_toggleFavorite`) for handling a tap. You’ll define the callback function next.

lib/main.dart (_FavoriteWidgetState build)

```
class _FavoriteWidgetState extends State<FavoriteWidget> {
  // ···
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        Container(
          padding: EdgeInsets.all(0),
          child: IconButton(
            icon: (_isFavorited ? Icon(Icons.star) :
Icon(Icons.star_border)),
            color: Colors.red[500],
            onPressed: _toggleFavorite,
          ),
        ),
        SizedBox(
          width: 18,
          child: Container(
            child: Text('$_favoriteCount'),
          ),
        ),
      ],
    );
  }
}
```

Actuce: Placing the `Text` in a SizedBox and setting its width prevents a discernible "jump" when the text changes between the values of 40 and 41 — a jump would otherwise occur because those values have different widths.

The `_toggleFavorite()` method, which is called when the `IconButton` is pressed, calls `setState()`. Calling `setState()`is critical, because this tells the framework that the widget's state has changed and that the widget should be redrawn. The function argument to `setState()` toggles the UI between these two states:

- A `star` icon and the number 41
- A `star_border` icon and the number 40

```
void _toggleFavorite() {
  setState(() {
    if (_isFavorited) {
      _favoriteCount -= 1;
      _isFavorited = false;
    } else {
      _favoriteCount += 1;
      _isFavorited = true;
    }
  });
}
```

Step 4: Plug the stateful widget into the widget tree

Add your custom stateful widget to the widget tree in the app's `build()` method. First, locate the code that creates the `Icon` and `Text`, and delete it. In the same location, create the stateful widget:

layout/lakes/{step6 → interactive}/lib/main.dart

```
@@ -10,2 +5,2 @@
10
    class MyApp extends StatelessWidget {
5
```

```
 @override
@@ -38,11 +33,7 @@
 ],
 ),
 ),
- ~~Icon~~(
+ FavoriteWidget(),
- Icons.star,
- color: Colors.red[500],
- ),
- Text('41'),
 ],
 ),
 );
@@ -117,3 +108,3 @@
 );
 }
```

```
  }
```

That’s it! When you hot reload the app, the star icon should now respond to taps.

Problems?

If you can’t get your code to run, look in your IDE for possible errors. Debugging Flutter Apps might help. If you still can’t find the problem, check your code against the interactive lakes example on GitHub.

- lib/main.dart
- pubspec.yaml
- lakes.jpg

If you still have questions, refer to any one of the developer community channels.

The rest of this page covers several ways a widget’s state can be managed, and lists other available interactive widgets.

Managing state

What's the point?

- There are different approaches for managing state.
- You, as the widget designer, choose which approach to use.
- If in doubt, start by managing state in the parent widget.

Who manages the stateful widget’s state? The widget itself? The parent widget? Both? Another object? The answer is… it depends. There are several valid ways to make your widget interactive. You, as the widget designer, make the decision based on how you expect your widget to be used. Here are the most common ways to manage state:

- The widget manages its own state
- The parent manages the widget’s state
- A mix-and-match approach

How do you decide which approach to use? The following principles should help you decide:

- If the state in question is user data, for example the checked or unchecked mode of a checkbox, or the position of a slider, then the state is best managed by the parent widget.
- If the state in question is aesthetic, for example an animation, then the state is best managed by the widget itself.

If in doubt, start by managing state in the parent widget.

We'll give examples of the different ways of managing state by creating three simple examples: TapboxA, TapboxB, and TapboxC. The examples all work similarly—each creates a container that, when tapped, toggles between a green or grey box. The `_active` boolean determines the color: green for active or grey for inactive.

These examples use GestureDetector to capture activity on the Container.

The widget manages its own state

Sometimes it makes the most sense for the widget to manage its state internally. For example, ListView automatically scrolls when its content exceeds the render box. Most developers using ListView don't want to manage ListView's scrolling behavior, so ListView itself manages its scroll offset.

The `_TapboxAState` class:

- Manages state for `TapboxA`.
- Defines the `_active` boolean which determines the box's current color.
- Defines the `_handleTap()` function, which updates `_active` when the box is tapped and calls the `setState()` function to update the UI.
- Implements all interactive behavior for the widget.

```
// TapboxA manages its own state.

//------------------------- TapboxA ----------------------
------------

class TapboxA extends StatefulWidget {
  TapboxA({Key key}) : super(key: key);

  @override
  _TapboxAState createState() => _TapboxAState();
}

class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            _active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color:
Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: _active ? Colors.lightGreen[700] :
Colors.grey[600],
        ),
      ),
    );
  }
}

//------------------------ MyApp ------------------------
----------

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

The parent widget manages the widget’s state

Often it makes the most sense for the parent widget to manage the state and tell its child widget when to update. For example, IconButton allows you to treat an icon as a tappable button. IconButton is a stateless widget because we decided that the parent widget needs to know whether the button has been tapped, so it can take appropriate action.

In the following example, TapboxB exports its state to its parent through a callback. Because TapboxB doesn’t manage any state, it subclasses StatelessWidget.

The ParentWidgetState class:

- Manages the `_active` state for TapboxB.
- Implements `_handleTapboxChanged()`, the method called when the box is tapped.
- When the state changes, calls `setState()` to update the UI.

The TapboxB class:

- Extends StatelessWidget because all state is handled by its parent.
- When a tap is detected, it notifies the parent.

```
// ParentWidget manages the state for TapboxB.

//------------------------ ParentWidget ------------------
--------------

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() =>
_ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: TapboxB(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}

//------------------------- TapboxB ----------------------
------------

class TapboxB extends StatelessWidget {
  TapboxB({Key key, this.active: false, @required
this.onChanged})
      : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  void _handleTap() {
    onChanged(!active);
  }

  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color:
Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: active ? Colors.lightGreen[700] :
Colors.grey[600],
        ),
      ),
    );
  }
}
```

Tip: When creating API, consider using the `@required` annotation for any parameters that your code relies on. To use `@required`, import the foundation library (which re-exports Dart's meta.dart library):

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

A mix-and-match approach

For some widgets, a mix-and-match approach makes the most sense. In this scenario, the stateful widget manages some of the state, and the parent widget manages other aspects of the state.

In the `TapboxC` example, on tap down, a dark green border appears around the box. On tap up, the border disappears and the box's color changes. `TapboxC` exports its `_active` state to its parent but manages its `_highlight` state internally. This example has two State objects, `_ParentWidgetState` and `_TapboxCState`.

The `_ParentWidgetState` object:

- Manages the `_active` state.
- Implements `_handleTapboxChanged()`, the method called when the box is tapped.
- Calls `setState()` to update the UI when a tap occurs and the `_active` state changes.

The `_TapboxCState` object:

- Manages the `_highlight` state.
- The `GestureDetector` listens to all tap events. As the user taps down, it adds the highlight (implemented as a dark green border). As the user releases the tap, it removes the highlight.
- Calls `setState()` to update the UI on tap down, tap up, or tap cancel, and the `_highlight` state changes.
- On a tap event, passes that state change to the parent widget to take appropriate action using the widget property.

```
//--------------------------- ParentWidget ----------------------------
---

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() =>
_ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: TapboxC(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}

//---------------------------- TapboxC ------------------
------------

class TapboxC extends StatefulWidget {
  TapboxC({Key key, this.active: false, @required
this.onChanged})
      : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  _TapboxCState createState() => _TapboxCState();
}

class _TapboxCState extends State<TapboxC> {
  bool _highlight = false;

  void _handleTapDown(TapDownDetails details) {
    setState(() {
      _highlight = true;
    });
  }

  void _handleTapUp(TapUpDetails details) {
    setState(() {
      _highlight = false;
    });
  }

  void _handleTapCancel() {
    setState(() {
      _highlight = false;
    });
  }

  void _handleTap() {
    widget.onChanged(!widget.active);
  }

  Widget build(BuildContext context) {
    // This example adds a green border on tap down.
    // On tap up, the square changes to the opposite
state.
    return GestureDetector(
      onTapDown: _handleTapDown, // Handle the tap events
in the order that
      onTapUp: _handleTapUp, // they occur: down, up, tap,
cancel
      onTap: _handleTap,
      onTapCancel: _handleTapCancel,
      child: Container(
        child: Center(
          child: Text(widget.active ? 'Active' :
'Inactive',
              style: TextStyle(fontSize: 32.0, color:
Colors.white)),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color:
              widget.active ? Colors.lightGreen[700] :
Colors.grey[600],
          border: _highlight
              ? Border.all(
                  color: Colors.teal[700],
                  width: 10.0,
                )
              : null,
        ),
      ),
    );
  }
}
```

An alternate implementation might have exported the highlight state to the parent while keeping the active state internal, but if you asked someone to use that tap box, they'd probably complain that it doesn't make much sense. The developer cares whether the box is active. The developer probably doesn't care how the highlighting is managed, and prefers that the tap box handles those details.

Other interactive widgets

Flutter offers a variety of buttons and similar interactive widgets. Most of these widgets implement the Material Design guidelines, which define a set of components with an opinionated UI.

If you prefer, you can use GestureDetector to build interactivity into any custom widget. You can find examples of GestureDetector in Managing state, and in the Flutter Gallery.

Tip: Flutter also provides a set of iOS-style widgets called Cupertino.

When you need interactivity, it's easiest to use one of the prefabricated widgets. Here's a partial list:

Standard widgets

- Form
- FormField

Material Components

- Checkbox
- DropdownButton
- FlatButton

- FloatingActionButton
- IconButton
- Radio
- RaisedButton
- Slider
- Switch
- TextField

Resources

The following resources might help when adding interactivity to your app.

- Gestures, a section in Introduction to widgets
  How to create a button and make it respond to input.
- Gestures in Flutter
  A description of Flutter's gesture mechanism.
- Flutter API documentation
  Reference documentation for all of the Flutter libraries.
- Flutter Gallery
  Demo app showcasing many Material Components and other Flutter features.
- Flutter's Layered Design (video)
  This video includes information about state and stateless widgets. Presented by Google engineer, Ian Hickson.

Checkbox class

A material design checkbox.

The checkbox itself does not maintain any state. Instead, when the state of the checkbox changes, the widget calls the onChanged callback. Most widgets that use a checkbox will listen for the onChangedcallback and rebuild the checkbox with a new value to update the visual appearance of the checkbox.

The checkbox can optionally display three values - true, false, and null - if tristate is true. When value is null a dash is displayed. By default tristate is false and the checkbox's value must be true or false.

Requires one of its ancestors to be a Material widget.

See also:

- CheckboxListTile, which combines this widget with a ListTile so that you can give the checkbox a label.
- Switch, a widget with semantics similar to Checkbox.
- Radio, for selecting among a set of explicit values.
- Slider, for selecting a value in a range.
- material.io/design/components/selection-controls.html#checkboxes
- material.io/design/components/lists.html#types

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- StatefulWidget
- Checkbox

Constructors

Checkbox({Key key, @required bool value, bool tristate: false, @required ValueChanged<bool> onChanged, Color activeColor, Color checkColor, MaterialTapTargetSize materialTapTargetSize })

Creates a material design checkbox. [...]

const

Properties

activeColor → Color

The color to use when this checkbox is checked. [...]

final

checkColor → Color

The color to use for the check icon when this checkbox is checked [...]

final

materialTapTargetSize → MaterialTapTargetSize

Configures the minimum size of the tap target. [...]

final

onChanged → ValueChanged<bool>

Called when the value of the checkbox should change. [...]

final

tristate → bool

If true the checkbox's value can be true, false, or null. [...]

final

value → bool

Whether this checkbox is checked. [...]

final

hashCode → int

The hash code for this object. [...]

read-only, inherited

key → Key

Controls how one widget replaces another widget in the tree. [...]

final, inherited

runtimeType → Type

A representation of the runtime type of the object.

read-only, inherited

Methods

createState() → _CheckboxState

Creates the mutable state for this widget at a given location in the tree. [...]

override

createElement() → StatefulElement

Creates a StatefulElement to manage this widget's location in the tree. [...]

inherited

debugDescribeChildren() → List<DiagnosticsNode>

Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → void

Add additional properties associated with the node. [...]

inherited

noSuchMethod(Invocation invocation) → dynamic

Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode

Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String

Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String

Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String

Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

Constants

width → const double
The width of a checkbox widget.

`18.0`

DropdownButton<T> class

A material design button for selecting from a list of items.

A dropdown button lets the user select from a number of items. The button shows the currently selected item as well as an arrow that opens a menu for selecting another item.

The type `T` is the type of the value that each dropdown item represents. All the entries in a given menu must represent values with consistent types. Typically, an enum is used. Each DropdownMenuItem in itemsmust be specialized with that same type argument.

The onChanged callback should update a state variable that defines the dropdown's value. It should also call State.setState to rebuild the dropdown with the new value.

SampleSample in an App

This sample shows a `DropdownButton` whose value is one of "One", "Two", "Free", or "Four".

```
String dropdownValue = 'One';

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: DropdownButton<String>(
        value: dropdownValue,
        onChanged: (String newValue) {
          setState(() {
            dropdownValue = newValue;
          });
        },
        items: <String>['One', 'Two', 'Free', 'Four']
          .map<DropdownMenuItem<String>>((String value) {
            return DropdownMenuItem<String>(
              value: value,
              child: Text(value),
            );
          })
          .toList(),
      ),
    ),
  );
}
```

If the onChanged callback is null or the list of items is null then the dropdown button will be disabled, i.e. its arrow will be displayed in grey and it will not respond to input. A disabled button will display the disabledHint widget if it is non-null.

Requires one of its ancestors to be a Material widget.

See also:

- DropdownMenuItem, the class used to represent the items.
- DropdownButtonHideUnderline, which prevents its descendant dropdown buttons from displaying their underlines.
- RaisedButton, FlatButton, ordinary buttons that trigger a single action.
- material.io/design/components/menus.html#dropdown-menu

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- StatefulWidget
- DropdownButton

Constructors

DropdownButton({Key key, @required List<DropdownMenuItem<T>> items, T value, Widget hint, Widget disabledHint, @required ValueChanged<T> onChanged, int elevation: 8, TextStyle style, Widget underline, Widget icon, Color iconDisabledColor, Color iconEnabledColor, double iconSize: 24.0, bool isDense: false, bool isExpanded: false})
    Creates a dropdown button. [...]

Properties

disabledHint → Widget
    A message to show when the dropdown is disabled. [...]

    final

elevation → int

The z-coordinate at which to place the menu when open. [...]

final

hint → Widget
Displayed if value is null.

final

icon → Widget
The widget to use for the drop-down button's icon. [...]

final

iconDisabledColor → Color
The color of any Icon descendant of icon if this button is disabled, i.e. if onChanged is null. [...]

final

iconEnabledColor → Color
The color of any Icon descendant of icon if this button is enabled, i.e. if onChanged is defined. [...]

final

iconSize → double
The size to use for the drop-down button's down arrow icon button. [...]

final

isDense → bool
Reduce the button's height. [...]

final

isExpanded → bool
Set the dropdown's inner contents to horizontally fill its parent. [...]

final

items → List<DropdownMenuItem<T>>
The list of items the user can select. [...]

final

onChanged → ValueChanged<T>
Called when the user selects an item. [...]

final

style → TextStyle

The text style to use for text in the dropdown button and the dropdown menu that appears when you tap the button. [...]

final

underline → Widget
The widget to use for drawing the drop-down button's underline. [...]

final

value → T
The value of the currently selected DropdownMenuItem, or null if no item has been selected. If `value` is null then the menu is popped up as if the first item were selected.

final

hashCode → int
The hash code for this object. [...]

read-only, inherited

key → Key
Controls how one widget replaces another widget in the tree. [...]

final, inherited

runtimeType → Type
A representation of the runtime type of the object.

read-only, inherited

Methods

createState() → _DropdownButtonState<T>
Creates the mutable state for this widget at a given location in the tree. [...]

override

createElement() → StatefulElement
Creates a StatefulElement to manage this widget's location in the tree. [...]

inherited

debugDescribeChildren() → List<DiagnosticsNode>
Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
Add additional properties associated with the node. [...]

inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode
Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

1. CONSTRUCTORS
2. DropdownButton
3. PROPERTIES
4. disabledHint
5. elevation
6. hint

7. icon
8. iconDisabledColor
9. iconEnabledColor
10. iconSize
11. isDense
12. isExpanded
13. items
14. onChanged
15. style
16. underline

FlatButton class

A material design "flat button".

A flat button is a text label displayed on a (zero elevation) Material widget that reacts to touches by filling with color.

Use flat buttons on toolbars, in dialogs, or inline with other content but offset from that content with padding so that the button's presence is obvious. Flat buttons intentionally do not have visible borders and must therefore rely on their position relative to other content for context. In dialogs and cards, they should be grouped together in one of the bottom corners. Avoid using flat buttons where they would blend in with other content, for example in the middle of lists.

Material design flat buttons have an all-caps label, some internal padding, and some defined dimensions. To have a part of your application be interactive, with ink splashes, without also committing to these stylistic choices, consider using InkWell instead.

If the onPressed callback is null, then the button will be disabled, will not react to touch, and will be colored as specified by the disabledColor property instead of the color property. If you are trying to change the button's color and it is not having any effect, check that you are passing a non-null onPressed handler.

Flat buttons have a minimum size of 88.0 by 36.0 which can be overridden with ButtonTheme.

The clipBehavior argument must not be null.

Sample

This example shows a simple FlatButton.

```
FlatButton(
  onPressed: () {
    /*...*/
  },
  child: Text(
    "Flat Button",
  ),
)
```

Sample

This example shows a FlatButton that is normally white-on-blue, with splashes rendered in a different shade of blue. It turns black-on-grey when disabled. The button has 8px of padding on each side, and the text is 20px high.

```
FlatButton(
  color: Colors.blue,
  textColor: Colors.white,
  disabledColor: Colors.grey,
  disabledTextColor: Colors.black,
  padding: EdgeInsets.all(8.0),
  splashColor: Colors.blueAccent,
  onPressed: () {
    /*...*/
  },
  child: Text(
    "Flat Button",
    style: TextStyle(fontSize: 20.0),
  ),
)
```

See also:

- RaisedButton, a filled button whose material elevates when pressed.
- DropdownButton, which offers the user a choice of a number of options.
- SimpleDialogOption, which is used in SimpleDialogs.
- IconButton, to create buttons that just contain icons.
- InkWell, which implements the ink splash part of a flat button.
- RawMaterialButton, the widget this widget is based on.
- material.io/design/components/buttons.html

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- StatelessWidget
- MaterialButton
- FlatButton

Constructors

FlatButton({Key key, @required VoidCallback onPressed, ValueChanged<bool> onHighlightChanged, ButtonTextTheme textTheme, Color textColor, Color disabledTextColor, Color color, Color disabledColor, Color focusColor, Color hoverColor, Color highlightColor, Color splashColor, Brightness colorBrightness, EdgeInsetsGeometry padding, ShapeBorder shape, Clip clipBehavior, FocusNode focusNode, MaterialTapTargetSize materialTapTargetSize, @required Widget child })

Create a simple text button.

const

FlatButton.icon({Key key, @required VoidCallback onPressed, ValueChanged<bool> onHighlightChanged, ButtonTextTheme textTheme, Color textColor, Color disabledTextColor, Color color, Color disabledColor, Color focusColor, Color hoverColor, Color highlightColor, Color splashColor, Brightness colorBrightness, EdgeInsetsGeometry padding, ShapeBorder shape, Clip clipBehavior, FocusNode focusNode, MaterialTapTargetSize materialTapTargetSize, @required Widget icon, @required Widget label })

Create a text button from a pair of widgets that serve as the button's `icon` and `label`. [...]

factory

Properties

animationDuration → Duration

Defines the duration of animated changes for shape and elevation. [...]

final, inherited

child → Widget

The button's label. [...]

final, inherited

clipBehavior → Clip

The content will be clipped (or not) according to this option. [...]

final, inherited

color → Color

The button's fill color, displayed by its Material, while it is in its default (unpressed, enabled) state. [...]

final, inherited

colorBrightness → Brightness

The theme brightness to use for this button. [...]

final, inherited

disabledColor → Color
The fill color of the button when the button is disabled. [...]

final, inherited

disabledElevation → double
The elevation for the button's Material relative to its parent when the button is not enabled. [...]

final, inherited

disabledTextColor → Color
The color to use for this button's text when the button is disabled. [...]

final, inherited

elevation → double
The z-coordinate at which to place this button relative to its parent. [...]

final, inherited

enabled → bool
Whether the button is enabled or disabled. [...]

read-only, inherited

focusColor → Color
The fill color of the button's Material when it has the input focus. [...]

final, inherited

focusElevation → double
The elevation for the button's Material when the button is enabled and has the input focus. [...]

final, inherited

focusNode → FocusNode
An optional focus node to use for requesting focus when pressed. [...]

final, inherited

hashCode → int
The hash code for this object. [...]

read-only, inherited

height → double
The vertical extent of the button. [...]

final, inherited

highlightColor → Color

The highlight color of the button's InkWell. [...]

final, inherited

highlightElevation → double
The elevation for the button's Material relative to its parent when the button is enabled and pressed. [...]

final, inherited

hoverColor → Color
The fill color of the button's Material when a pointer is hovering over it. [...]

final, inherited

hoverElevation → double
The elevation for the button's Material when the button is enabled and a pointer is hovering over it. [...]

final, inherited

key → Key
Controls how one widget replaces another widget in the tree. [...]

final, inherited

materialTapTargetSize → MaterialTapTargetSize
Configures the minimum size of the tap target. [...]

final, inherited

minWidth → double
The smallest horizontal extent that the button will occupy. [...]

final, inherited

onHighlightChanged → ValueChanged<bool>
Called by the underlying InkWell widget's InkWell.onHighlightChanged callback. [...]

final, inherited

onPressed → VoidCallback
The callback that is called when the button is tapped or otherwise activated. [...]

final, inherited

padding → EdgeInsetsGeometry
The internal padding for the button's child. [...]

final, inherited

runtimeType → Type
A representation of the runtime type of the object.

read-only, inherited

shape → ShapeBorder
The shape of the button's Material. [...]

final, inherited

splashColor → Color
The splash color of the button's InkWell. [...]

final, inherited

textColor → Color
The color to use for this button's text. [...]

final, inherited

textTheme → ButtonTextTheme
Defines the button's base colors, and the defaults for the button's minimum size, internal padding, and shape. [...]

final, inherited

Methods

build(BuildContext context) → Widget
Describes the part of the user interface represented by this widget. [...]

override

createElement() → StatelessElement
Creates a StatelessElement to manage this widget's location in the tree. [...]

inherited

debugDescribeChildren() → List<DiagnosticsNode>
Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
Add additional properties associated with the node. [...]

inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode

Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

FloatingActionButton class

A material design floating action button.

A floating action button is a circular icon button that hovers over content to promote a primary action in the application. Floating action buttons are most commonly used in the Scaffold.floatingActionButton field.

Use at most a single floating action button per screen. Floating action buttons should be used for positive actions such as "create", "share", or "navigate". (If more than one floating action button is used within aRoute, then make sure that each button has a unique heroTag, otherwise an exception will be thrown.)

If the onPressed callback is null, then the button will be disabled and will not react to touch. It is highly discouraged to disable a floating action button as there is no indication to the user that the button is disabled. Consider changing the backgroundColor if disabling the floating action button.

SampleSample in an App

This example shows how to make a simple FloatingActionButton in a Scaffold, with a pink backgroundColor and a thumbs up Icon.

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Floating Action Button Sample'),
    ),
    body: Center(
      child: Text('Press the button below!')
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        // Add your onPressed code here!
      },
      child: Icon(Icons.thumb_up),
      backgroundColor: Colors.pink,
    ),
  );
}
```

SampleSample in an App

This example shows how to make an extended FloatingActionButton in a Scaffold, with a pink backgroundColor and a thumbs up Icon and a Text label.

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Floating Action Button Sample'),
    ),
    body: Center(
      child: Text('Press the extended button below!'),
    ),
    floatingActionButton: FloatingActionButton.extended(
      onPressed: () {
        // Add your onPressed code here!
      },
      label: Text('Approve'),
      icon: Icon(Icons.thumb_up),
      backgroundColor: Colors.pink,
    ),
  );
}
```

See also:

- Scaffold, in which floating action buttons typically live.
- RaisedButton, another kind of button that appears to float above the content.
- material.io/design/components/buttons-floating-action-button.html

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- StatelessWidget
- FloatingActionButton

Constructors

FloatingActionButton({Key key, Widget child, String tooltip, Color foregroundColor, Color backgroundColor, Color focusColor, Color hoverColor, Object heroTag: const _DefaultHeroTag(), double elevation, double focusElevation, double hoverElevation, double highlightElevation, double disabledElevation, @required VoidCallback onPressed, bool mini: false, ShapeBorder shape, Clip clipBehavior: Clip.none, FocusNode focusNode, MaterialTapTargetSize materialTapTargetSize, bool isExtended: false })

Creates a circular floating action button. [...]

const

FloatingActionButton.extended({Key key, String tooltip, Color foregroundColor, Color backgroundColor, Color focusColor, Color hoverColor, Object heroTag: const _DefaultHeroTag(), double elevation, double focusElevation, double hoverElevation, double highlightElevation, double disabledElevation, @required VoidCallback onPressed, ShapeBorder shape, bool isExtended: true, MaterialTapTargetSize materialTapTargetSize, Clip clipBehavior: Clip.none, FocusNode focusNode, Widget icon, @required Widget label })

Creates a wider StadiumBorder-shaped floating action button with an optional `icon` and a `label`. [...]

Properties

backgroundColor → Color

The color to use when filling the button. [...]

final

child → Widget

The widget below this widget in the tree. [...]

final

clipBehavior → Clip

The content will be clipped (or not) according to this option. [...]

final

disabledElevation → double

The z-coordinate at which to place this button when the button is disabled (onPressed is null). [...]

final

elevation → double

The z-coordinate at which to place this button relative to its parent. [...]

final

focusColor → Color

The color to use for filling the button when the button has input focus. [...]

final

focusElevation → double
The z-coordinate at which to place this button relative to its parent when the button has the input focus. [...]

final

focusNode → FocusNode
An optional focus node to use for requesting focus when pressed. [...]

final

foregroundColor → Color
The default icon and text color. [...]

final

heroTag → Object
The tag to apply to the button's Hero widget. [...]

final

highlightElevation → double
The z-coordinate at which to place this button relative to its parent when the user is touching the button. [...]

final

hoverColor → Color
The color to use for filling the button when the button has a pointer hovering over it. [...]

final

hoverElevation → double
The z-coordinate at which to place this button relative to its parent when the button is enabled and has a pointer hovering over it. [...]

final

isExtended → bool
True if this is an "extended" floating action button. [...]

final

materialTapTargetSize → MaterialTapTargetSize
Configures the minimum size of the tap target. [...]

final

mini → bool
Controls the size of this button. [...]

final

onPressed → VoidCallback
    The callback that is called when the button is tapped or otherwise activated. [...]

    final

shape → ShapeBorder
    The shape of the button's Material. [...]

    final

tooltip → String
    Text that describes the action that will occur when the button is pressed. [...]

    final

hashCode → int
    The hash code for this object. [...]

    read-only, inherited

key → Key
    Controls how one widget replaces another widget in the tree. [...]

    final, inherited

runtimeType → Type
    A representation of the runtime type of the object.

    read-only, inherited

Methods

build(BuildContext context) → Widget
    Describes the part of the user interface represented by this widget. [...]

    override

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
    Add additional properties associated with the node. [...]

    override

createElement() → StatelessElement
    Creates a StatelessElement to manage this widget's location in the tree. [...]

    inherited

debugDescribeChildren() → List<DiagnosticsNode>
    Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode
Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

Radio<T> class

A material design radio button.

Used to select between a number of mutually exclusive values. When one radio button in a group is selected, the other radio buttons in the group cease to be selected. The values are of type `T`, the type parameter of the Radio class. Enums are commonly used for this purpose.

The radio button itself does not maintain any state. Instead, selecting the radio invokes the onChangedcallback, passing value as a parameter. If groupValue and value match, this radio will be selected. Most widgets will respond to onChanged by calling State.setState to update the radio button's groupValue.

SampleSample in an App

Here is an example of Radio widgets wrapped in ListTiles, which is similar to what you could get with the RadioListTile widget.
The currently selected character is passed into `groupValue`, which is maintained by the example's `State`. In this case, the first `Radio` will start off selected because `_character` is initialized to`SingingCharacter.lafayette`.

If the second radio button is pressed, the example's state is updated with `setState`, updating `_character` to `SingingCharacter.jefferson`. This causes the buttons to rebuild with the updated `groupValue`, and therefore the selection of the second button.

Requires one of its ancestors to be a Material widget.

```
enum SingingCharacter { lafayette, jefferson }

// ...

SingingCharacter _character = SingingCharacter.lafayette;

Widget build(BuildContext context) {
  return Center(
    child: Column(
      children: <Widget>[
        ListTile(
          title: const Text('Lafayette'),
          leading: Radio(
            value: SingingCharacter.lafayette,
            groupValue: _character,
            onChanged: (SingingCharacter value) {
              setState(() { _character = value; });
            },
          ),
        ),
        ListTile(
          title: const Text('Thomas Jefferson'),
          leading: Radio(
            value: SingingCharacter.jefferson,
            groupValue: _character,
            onChanged: (SingingCharacter value) {
              setState(() { _character = value; });
            },
          ),
        ),
      ],
    ),
  );
}
```

See also:

- RadioListTile, which combines this widget with a ListTile so that you can give the radio button a label.
- Slider, for selecting a value in a range.
- Checkbox and Switch, for toggling a particular value on or off.
- material.io/design/components/selection-controls.html#radio-buttons

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- StatefulWidget
- Radio

Constructors

Radio({Key key, @required T value, @required T groupValue, @required ValueChanged<T > onChanged, Color activeColor, MaterialTapTargetSize materialTapTargetSize })

Creates a material design radio button. [...]

const

Properties

activeColor → Color
: The color to use when this radio button is selected. [...]

final

groupValue → T
: The currently selected value for a group of radio buttons. [...]

final

materialTapTargetSize → MaterialTapTargetSize
: Configures the minimum size of the tap target. [...]

final

onChanged → ValueChanged<T>
: Called when the user selects this radio button. [...]

final

value → T
: The value represented by this radio button.

final

hashCode → int
: The hash code for this object. [...]

read-only, inherited

key → Key
: Controls how one widget replaces another widget in the tree. [...]

final, inherited

runtimeType → Type
: A representation of the runtime type of the object.

read-only, inherited

Methods

createState() → _RadioState<T>
: Creates the mutable state for this widget at a given location in the tree. [...]

override

createElement() → StatefulElement
Creates a StatefulElement to manage this widget's location in the tree. [...]

inherited

debugDescribeChildren() → List<DiagnosticsNode>
Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
Add additional properties associated with the node. [...]

inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode
Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool

The equality operator. [...]

inherited

AQ

introduction

Qu'est-ce que Flutter?

Flutter est la boîte à outils d’interface utilisateur portable de Google qui permet de créer de superbes applications natives pour les mobiles, le Web et les ordinateurs de bureau, à partir d’une seule base de code. Flutter fonctionne avec le code existant, est utilisé par les développeurs et les organisations du monde entier. Il est gratuit et à code source ouvert.

Que fait Flutter?

Pour les utilisateurs, Flutter donne vie à de belles interfaces utilisateur.

Pour les développeurs, Flutter abaisse la barre d'accès à la création d'applications mobiles. Il accélère le développement d'applications mobiles et réduit les coûts et la complexité de la production d'applications sur plusieurs plates-formes.

Pour les concepteurs, Flutter contribue à concrétiser la vision de conception originale, sans perte de fidélité ni compromis. C'est aussi un outil de prototypage productif.

Pour qui est-ce que Flutter?

Flutter est destiné aux développeurs qui recherchent un moyen plus rapide de créer de superbes applications mobiles ou un moyen de toucher davantage d'utilisateurs avec un seul investissement.

Flutter est également destiné aux responsables techniques qui doivent diriger des équipes de développement mobiles. Flutter permet aux responsables techniques de créer une seule équipe de développement d'applications mobiles, en unifiant leurs investissements de développement pour expédier plus de fonctionnalités plus rapidement, expédier le même jeu de fonctionnalités simultanément sur iOS et Android et réduire les coûts de maintenance.

Bien que ce ne soit pas le public cible initial, Flutter s'adresse également aux concepteurs qui souhaitent que leurs visions de conception d'origine soient livrées de manière cohérente, avec une grande fidélité, à tous les utilisateurs du mobile.

Fondamentalement, Flutter est destiné aux utilisateurs qui recherchent de superbes applications, avec une animation et un mouvement délicieux, et des interfaces utilisateur personnalisées avec une identité et une identité propres.

Quelle expérience dois-je avoir d'un programmeur / développeur pour utiliser Flutter?

Flutter est accessible aux programmeurs familiarisés avec les concepts orientés objet (classes, méthodes, variables, etc.) et les concepts de programmation impératifs (boucles, conditions, etc.).

Aucune expérience mobile préalable n'est requise pour apprendre et utiliser Flutter.

Nous avons vu des personnes ayant très peu d'expérience en programmation apprendre et utiliser Flutter pour le prototypage et le développement d'applications.

Quels types d'applications puis-je créer avec Flutter?

Flutter est optimisé pour les applications mobiles 2D qui veulent s'exécuter sur Android et iOS.

Les applications qui doivent proposer des conceptions originales sont particulièrement bien adaptées à Flutter. Toutefois, les applications qui doivent ressembler à des applications de plate-forme standard peuvent également être créées avec Flutter.

Vous pouvez créer des applications complètes avec Flutter, notamment une caméra, une géolocalisation, un réseau, un stockage, des kits de développement logiciel (SDK) tiers, etc.

Qui fait Flutter?

Flutter est un projet open source, avec des contributions de Google et de la communauté.

Qui utilise Flutter?

Les développeurs à l'intérieur et à l'extérieur de Google utilisent Flutter pour créer de superbes applications natives pour iOS et Android. Pour en savoir plus sur certaines de ces applications, visitez la vitrine.

Qu'est-ce qui rend Flutter unique?

Flutter est différent de la plupart des autres options de création d'applications mobiles, car Flutter n'utilise ni WebView ni les widgets OEM fournis avec le périphérique. Au lieu de cela, Flutter utilise son propre moteur de rendu hautes performances pour dessiner des widgets.

De plus, Flutter est différent car il ne comporte qu'une fine couche de code C / C ++. Flutter implémente la plupart de ses systèmes (composition, gestes, animation, framework, widgets, etc.) en Dart (langage moderne, concis, orienté objet) que les développeurs peuvent facilement aborder en lecture, modification, remplacement ou suppression. Cela donne aux développeurs un contrôle considérable sur le système et réduit considérablement la barre d'accès pour la majorité des systèmes.

Devrais-je créer ma prochaine application de production avec Flutter?

Flutter 1.0 a été lancé le 4 décembre 2018. Des milliers d'applications ont été livrées avec Flutter à des centaines de millions d'appareils. Voir quelques exemples d'applications dans la vitrine.

Pour plus d'informations sur le lancement et les versions ultérieures, voir Flutter 1.0: Boîte à outils de l'interface utilisateur portable de Google.

Qu'est-ce que Flutter fournit?

Qu'y a-t-il à l'intérieur du SDK Flutter?

Moteur de rendu 2D mobile, hautement optimisé, offrant une excellente prise en charge du texte

Cadre moderne de type réaction

Rich ensemble de widgets pour Android et iOS

API pour les tests unitaires et d'intégration

API d'interopérabilité et de plug-in pour la connexion au SDK système et tiers

Testeur sans tête pour l'exécution de tests sous Windows, Linux et Mac

Outils de ligne de commande pour créer, créer, tester et compiler vos applications

Flutter fonctionne-t-il avec des éditeurs ou des IDE?

Nous supportons les plugins pour Android Studio, IntelliJ IDEA et VS Code.

Reportez-vous à la configuration de l'éditeur pour obtenir des détails sur la configuration, ainsi qu'à Android Studio / IntelliJ et VS Code pour obtenir des conseils sur l'utilisation des plug-ins.

Vous pouvez également utiliser une combinaison de la commande Flutter dans un terminal et de l'un des nombreux éditeurs prenant en charge l'édition Dart.

Est-ce que Flutter vient avec un cadre?

Oui! Flutter est livré avec un cadre moderne, inspiré de React. La structure de Flutter est conçue pour être superposée et personnalisable (et facultative). Les développeurs peuvent choisir de n'utiliser que des parties du cadre ou un cadre différent.

Est-ce que Flutter est livré avec des widgets?

Oui! Flutter est livré avec un ensemble de widgets, de dispositions et de thèmes Cupercino (style iOS) de haute qualité. Bien sûr, ces widgets sont sur

Envoyer des commentaires

Historique

Enregistré

Communauté

Documentation Flutter

Lancez-vous

Configurez vous environnement de développement et commencer à compiler.

Catalogue de Widgets

Plonger dans le catalogue des widgets Flutter disponibles dans le SDK

Documentation API

Visitez l'API de référence de l'environnement Flutter

Le livres des tutos

Browse the cookbook for many easy Flutter recipes.

Samples

Check out the Flutter examples.

Videos

View the many videos on the Flutter Youtube channel.

Catalogue de Wigets

1. Documentation
2. Développement
3. Interface utilisateur
4. Widgets

Créez de superbes applications plus rapidement avec la collection de widgets visuels, structurels et interactifs. En plus de parcourir les widgets par catégorie, vous pouvez également voir tous les widgets dans le widget index.

Accessibility

Rendez votre application accessible.

Accessibility widgets

Make your app accessible.

See more widgets in the widget catalog.

ExcludeSemantics

A widget that drops all the semantics of its descendants. This can be used to hide subwidgets that would otherwise be reported but that would only be confusing. For example, the Material Components Chip widget hides the avatar since it is redundant with the chip label.

xcludeSemantics

Un widget qui supprime toute la sémantique de ses descendants. Cela peut être utilisé pour masquer des sous-widgets qui seraient autrement signalés mais qui ne feraient que dérouter. Par exemple, le widget Matière Composants masque l’avatar car il est redondant avec l’étiquette de la puce.

ExcludeSemantics class

A widget that drops all the semantics of its descendants.

When excluding is true, this widget (and its subtree) is excluded from the semantics tree.

This can be used to hide descendant widgets that would otherwise be reported but that would only be confusing. For example, the material library's Chip widget hides the avatar since it is redundant with the chip label.

See also:

- BlockSemantics which drops semantics of widgets earlier in the tree.

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget
- RenderObjectWidget
- SingleChildRenderObjectWidget
- ExcludeSemantics

Constructors

ExcludeSemantics({Key key, bool excluding: true, Widget child })
Creates a widget that drops all the semantics of its descendants.

const

Properties

excluding → bool
Whether this widget is excluded in the semantics tree.

final

child → Widget

The widget below this widget in the tree. [...]

final, inherited

hashCode → int
The hash code for this object. [...]

read-only, inherited

key → Key
Controls how one widget replaces another widget in the tree. [...]

final, inherited

runtimeType → Type
A representation of the runtime type of the object.

read-only, inherited

Methods

createRenderObject(BuildContext context) → RenderExcludeSemantics
Creates an instance of the RenderObject class that this RenderObjectWidget represents, using the configuration described by this RenderObjectWidget. [...]

override

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
Add additional properties associated with the node. [...]

override

updateRenderObject(BuildContext context, covariant RenderExcludeSemantics renderObject) → void
Copies the configuration described by this RenderObjectWidget to the given RenderObject, which will be of the same type as returned by this object's createRenderObject. [...]

override

createElement() → SingleChildRenderObjectElement
RenderObjectWidgets always inflate to a RenderObjectElement subclass.

inherited

debugDescribeChildren() → List<DiagnosticsNode>
Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

didUnmountRenderObject(covariant RenderObject renderObject) → void
A render object previously associated with this widget has been removed from the tree. The given RenderObject will be of the same type as returned by this object's createRenderObject.

@protected, inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode
Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

Classe ExcludeSemantics

Un widget qui supprime toute la sémantique de ses descendants.

Lorsque exclut est vrai, ce widget (et sa sous-arborescence) est exclu de l'arbre sémantique.

Cela peut être utilisé pour masquer les widgets descendants qui seraient autrement signalés mais qui ne feraient que dérouter. Par exemple, le widget Puce de la bibliothèque de matériaux masque l'avatar car il est redondant avec l'étiquette de la puce.

Voir également:

BlockSemantics qui supprime la sémantique des widgets plus tôt dans l’arborescence.

Héritage

Objet Diagnosticable DiagnosticableTree Widget RenderObjectWidget SingleChildRenderObjectWidget ExcludeSemantics

Constructeurs

ExcludeSemantics ({Key key, bool except: true, Widget child})

Crée un widget qui supprime toute la sémantique de ses descendants.

const

Propriétés

à l'exclusion → bool

Si ce widget est exclu dans l'arborescence sémantique.

final

enfant → widget

Le widget situé sous ce widget dans l'arborescence. [...]

final, hérité

hashCode → int

Le code de hachage pour cet objet. [...]

en lecture seule, hérité

clé → clé

Contrôle la manière dont un widget remplace un autre widget dans l'arborescence. [...]

final, hérité

runtimeType → Type

Une représentation du type d'exécution de l'objet.

en lecture seule, hérité

Les méthodes

createRenderObject (contexte BuildContext) → RenderExcludeSemantics

Crée une instance de la classe RenderObject que ce RenderObjectWidget représente, à l'aide de la configuration décrite par ce RenderObjectWidget. [...]

passer outre

debugFillProperties (propriétés de DiagnosticPropertiesBuilder) → void

Ajoutez des propriétés supplémentaires associées au nœud. [...]

passer outre

updateRenderObject (contexte BuildContext, covariant RenderExcludeSemantics renderObject) → void

Copie la configuration décrite par ce RenderObjectWidget dans le RenderObject donné, qui sera du même type que celui renvoyé par createRenderObject de cet objet. [...]

passer outre

createElement () → SingleChildRenderObjectElement

RenderObjectWidgets se gonfle toujours dans une sous-classe RenderObjectElement.

hérité

debugDescribeChildren () → Liste <DiagnosticsNode>

Renvoie une liste d'objets DiagnosticsNode décrivant les enfants de ce nœud. [...]

@protected, hérité

didUnmountRenderObject (RenderObject covariant renderObject) → void

Un objet de rendu précédemment associé à ce widget a été supprimé de l'arborescence. Le RenderObject donné sera du même type que celui renvoyé par createRenderObject de cet objet.

@protected, hérité

noSuchMethod (invocation d'invocation) → dynamique

Appelé lors de l'accès à une méthode ou à une propriété non existante. [...]

hérité

toDiagnosticsNode ({Nom de la chaîne, style DiagnosticsTreeStyle}) → DiagnosticsNode

Renvoie une représentation de débogage de l'objet utilisée par les outils de débogage et par DiagnosticsNode.toStringDeep. [...]

hérité

toString ({DiagnosticLevel minLevel: DiagnosticLevel.debug}) → Chaîne

Retourne une représentation sous forme de chaîne de cet objet.

hérité

toStringDeep ({String prefixLineOne: '', Préfixe de chaîneOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug}) → String

Retourne une représentation sous forme de chaîne de ce noeud et de ses descendants. [...]

hérité

toStringShallow ({menuisier de chaîne: ',', DiagnosticLevel minLevel: DiagnosticLevel.debug}) → Chaîne

Renvoie une description détaillée d'une ligne de l'objet. [...]

hérité

toStringShort () → Chaîne

Une courte description textuelle de ce widget.

hérité

Les opérateurs

opérateur == (autre dynamique) → bool

L'opérateur d'égalité. [...]

hérité

MergeSemantics

A widget that merges the semantics of its descendants.

FusionnerSémantique

Un widget qui fusionne la sémantique de ses descendants.

MergeSemantics class

A widget that merges the semantics of its descendants.

Causes all the semantics of the subtree rooted at this node to be merged into one node in the semantics tree. For example, if you have a widget with a Text node next to a checkbox widget, this could be used to merge the label from the Text node with the "checked" semantic state of the checkbox into a single node that had both the label and the checked state. Otherwise, the label would be presented as a separate feature than the checkbox, and the user would not be able to be sure that they were related.

Be aware that if two nodes in the subtree have conflicting semantics, the result may be nonsensical. For example, a subtree with a checked checkbox and an unchecked checkbox will be presented as checked. All the labels will be merged into a single string (with newlines separating each label from the other). If multiple nodes in the merged subtree can handle semantic gestures, the first one in tree order will be the one to receive the callbacks.

Inheritance

- Object

- Diagnosticable
- DiagnosticableTree
- Widget
- RenderObjectWidget
- SingleChildRenderObjectWidget
- MergeSemantics

Constructors

MergeSemantics({Key key, Widget child })

Creates a widget that merges the semantics of its descendants.

const

Properties

child → Widget

The widget below this widget in the tree. [...]

final, inherited

hashCode → int

The hash code for this object. [...]

read-only, inherited

key → Key

Controls how one widget replaces another widget in the tree. [...]

final, inherited

runtimeType → Type

A representation of the runtime type of the object.

read-only, inherited

Methods

createRenderObject(BuildContext context) → RenderMergeSemantics

Creates an instance of the RenderObject class that this RenderObjectWidget represents, using the configuration described by this RenderObjectWidget. [...]

override

createElement() → SingleChildRenderObjectElement

RenderObjectWidgets always inflate to a RenderObjectElement subclass.

inherited

debugDescribeChildren() → List<DiagnosticsNode>

Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → void

Add additional properties associated with the node. [...]

inherited

didUnmountRenderObject(covariant RenderObject renderObject) → void

A render object previously associated with this widget has been removed from the tree. The given RenderObject will be of the same type as returned by this object's createRenderObject.

@protected, inherited

noSuchMethod(Invocation invocation) → dynamic

Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode

Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String

Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String

Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

updateRenderObject(BuildContext context, covariant RenderObject renderObject) → void
Copies the configuration described by this RenderObjectWidget to the given RenderObject, which will be of the same type as returned by this object's createRenderObject. [...]

@protected, inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

Classe MergeSemantics

Un widget qui fusionne la sémantique de ses descendants.

Entraîne la fusion de toutes les sémantiques du sous-arbre enraciné sur ce nœud en un seul nœud de l'arborescence. Par exemple, si vous avez un widget avec un noeud Texte à côté d'un widget à case à cocher, vous pouvez utiliser cette étiquette pour fusionner l'étiquette du noeud Text avec l'état sémantique "coché" de la case à cocher dans un seul noeud doté à la fois de l'étiquette et du symbole. l'état vérifié. Sinon, l'étiquette serait présentée comme une fonctionnalité distincte de la case à cocher, et l'utilisateur ne pourrait pas être sûr de leur lien.

Sachez que si deux nœuds du sous-arbre ont des sémantiques en conflit, le résultat risque d'être absurde. Par exemple, une sous-arborescence avec une case à cocher cochée et une case à cocher non cochée sera présentée comme étant cochée. Toutes les étiquettes seront fusionnées en une seule chaîne (avec de nouvelles lignes séparant chaque étiquette de l'autre). Si plusieurs nœuds du sous-arbre fusionné peuvent gérer des gestes sémantiques, le premier dans l’arbre sera celui qui recevra les rappels.

Héritage

Objet Diagnosticable DiagnosticableTree Widget RenderObjectWidget SingleChildRenderObjectWidget MergeSemantics

Constructeurs

MergeSemantics ({Key key, Widget child})

Crée un widget qui fusionne la sémantique de ses descendants.

const

Propriétés

enfant → widget

Le widget situé sous ce widget dans l'arborescence. [...]

final, hérité

hashCode → int

Le code de hachage pour cet objet. [...]

en lecture seule, hérité

clé → clé

Contrôle la manière dont un widget remplace un autre widget dans l'arborescence. [...]

final, hérité

runtimeType → Type

Une représentation du type d'exécution de l'objet.

en lecture seule, hérité

Les méthodes

createRenderObject (contexte BuildContext) → RenderMergeSemantics

Crée une instance de la classe RenderObject que ce RenderObjectWidget représente, à l'aide de la configuration décrite par ce RenderObjectWidget. [...]

passer outre

createElement () → SingleChildRenderObjectElement

RenderObjectWidgets se gonfle toujours dans une sous-classe RenderObjectElement.

hérité

debugDescribeChildren () → Liste <DiagnosticsNode>

Renvoie une liste d'objets DiagnosticsNode décrivant les enfants de ce nœud. [...]

@protected, hérité

debugFillProperties (propriétés de DiagnosticPropertiesBuilder) → void

Ajoutez des propriétés supplémentaires associées au nœud. [...]

hérité

didUnmountRenderObject (RenderObject covariant renderObject) → void

Un objet de rendu précédemment associé à ce widget a été supprimé de l'arborescence. Le RenderObject donné sera du même type que celui renvoyé par createRenderObject de cet objet.

@protected, hérité

noSuchMethod (invocation d'invocation) → dynamique

Appelé lors de l'accès à une méthode ou à une propriété non existante. [...]

hérité

toDiagnosticsNode ({Nom de la chaîne, style DiagnosticsTreeStyle}) → DiagnosticsNode

Renvoie une représentation de débogage de l'objet utilisée par les outils de débogage et par DiagnosticsNode.toStringDeep. [...]

hérité

toString ({DiagnosticLevel minLevel: DiagnosticLevel.debug}) → Chaîne

Retourne une représentation sous forme de chaîne de cet objet.

hérité

toStringDeep ({String prefixLineOne: '', Préfixe de chaîneOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug}) → String

Retourne une représentation sous forme de chaîne de ce noeud et de ses descendants. [...]

hérité

toStringShallow ({menuisier de chaîne: ',', DiagnosticLevel minLevel: DiagnosticLevel.debug}) → Chaîne

Renvoie une description détaillée d'une ligne de l'objet. [...]

hérité

toStringShort () → Chaîne

Une courte description textuelle de ce widget.

hérité

updateRenderObject (contexte BuildContext, RenderObject renderObject covariant) → void

Copie la configuration décrite par ce RenderObjectWidget dans le RenderObject donné, qui sera du même type que celui renvoyé par createRenderObject de cet objet. [...]

@protected, hérité

Les opérateurs

opérateur == (autre dynamique) → bool

L'opérateur d'égalité. [...]

hérité

Semantics

A widget that annotates the widget tree with a description of the meaning of the widgets. Used by accessibility tools, search engines, and other semantic analysis software to determine the meaning of the application.

Sémantique

Un widget qui annote l’arborescence des widgets avec une description de la signification des widgets. Utilisé par les outils d'accessibilité, les moteurs de recherche et d'autres logiciels d'analyse sémantique pour déterminer la signification de l'application.

Semantics class

A widget that annotates the widget tree with a description of the meaning of the widgets.

Used by accessibility tools, search engines, and other semantic analysis software to determine the meaning of the application.

See also:

- MergeSemantics, which marks a subtree as being a single node for accessibility purposes.
- ExcludeSemantics, which excludes a subtree from the semantics tree (which might be useful if it is, e.g., totally decorative and not important to the user).
- `RenderObject.semanticsAnnotator`, the rendering library API through which the Semanticswidget is actually implemented.
- SemanticsNode, the object used by the rendering library to represent semantics in the semantics tree.
- SemanticsDebugger, an overlay to help visualize the semantics tree. Can be enabled using WidgetsApp.showSemanticsDebugger or MaterialApp.showSemanticsDebugger .

Inheritance

- Object
- Diagnosticable
- DiagnosticableTree
- Widget

- RenderObjectWidget

- SingleChildRenderObjectWidget
- Semantics

Annotations

  - @immutable

Constructors

Semantics({Key key, Widget child, bool container: false, bool explicitChildNodes: false, bool excludeSemantics: false, bool enabled, bool checked, bool selected, bool toggled, bool button, bool header, bool textField, bool readOnly, bool focused, bool inMutuallyExclusiveGroup, bool obscured, bool scopesRoute, bool namesRoute, bool hidden, bool image, bool liveRegion, String label, String value, String increasedValue, String decreasedValue, String hint, String onTapHint, String onLongPressHint, TextDirection textDirection, SemanticsSortKey sortKey, VoidCallback onTap, VoidCallback onLongPress, VoidCallback onScrollLeft, VoidCallback onScrollRight, VoidCallback onScrollUp, VoidCallback onScrollDown, VoidCallback onIncrease, VoidCallback onDecrease, VoidCallback onCopy, VoidCallback onCut, VoidCallback onPaste, VoidCallback onDismiss, MoveCursorHandler onMoveCursorForwardByCharacter, MoveCursorHandler onMoveCursorBackwardByCharacter, SetSelectionHandler onSetSelection, VoidCallback onDidGainAccessibilityFocus, VoidCallback onDidLoseAccessibilityFocus, Map<CustomSemanticsAction, VoidCallback> customSemanticsActions })
    Creates a semantic annotation. [...]

Semantics.fromProperties({Key key, Widget child, bool container: false, bool explicitChildNodes: false, bool excludeSemantics: false, @required SemanticsProperties properties })
    Creates a semantic annotation using SemanticsProperties. [...]

    const

Properties

container → bool
    If container is true, this widget will introduce a new node in the semantics tree. Otherwise, the semantics will be merged with the semantics of any ancestors (if the ancestor allows that). [...]

    final

excludeSemantics → bool
    Whether to replace all child semantics with this node. [...]

final

explicitChildNodes → bool
: Whether descendants of this widget are allowed to add semantic information to the SemanticsNode annotated by this widget. [...]

  final

properties → SemanticsProperties
: Contains properties used by assistive technologies to make the application more accessible.

  final

child → Widget
: The widget below this widget in the tree. [...]

  final, inherited

hashCode → int
: The hash code for this object. [...]

  read-only, inherited

key → Key
: Controls how one widget replaces another widget in the tree. [...]

  final, inherited

runtimeType → Type
: A representation of the runtime type of the object.

  read-only, inherited

Methods

createRenderObject(BuildContext context) → RenderSemanticsAnnotations
: Creates an instance of the RenderObject class that this RenderObjectWidget represents, using the configuration described by this RenderObjectWidget. [...]

  override

debugFillProperties(DiagnosticPropertiesBuilder properties) → void
: Add additional properties associated with the node. [...]

  override

updateRenderObject(BuildContext context, covariant RenderSemanticsAnnotations renderObject) → void

Copies the configuration described by this RenderObjectWidget to the given RenderObject, which will be of the same type as returned by this object's createRenderObject. [...]

override

createElement() → SingleChildRenderObjectElement
RenderObjectWidgets always inflate to a RenderObjectElement subclass.

inherited

debugDescribeChildren() → List<DiagnosticsNode>
Returns a list of DiagnosticsNode objects describing this node's children. [...]

@protected, inherited

didUnmountRenderObject(covariant RenderObject renderObject) → void
A render object previously associated with this widget has been removed from the tree. The given RenderObject will be of the same type as returned by this object's createRenderObject.

@protected, inherited

noSuchMethod(Invocation invocation) → dynamic
Invoked when a non-existent method or property is accessed. [...]

inherited

toDiagnosticsNode({String name, DiagnosticsTreeStyle style }) → DiagnosticsNode
Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep. [...]

inherited

toString({DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a string representation of this object.

inherited

toStringDeep({String prefixLineOne: '', String prefixOtherLines, DiagnosticLevel minLevel: DiagnosticLevel.debug })→ String
Returns a string representation of this node and its descendants. [...]

inherited

toStringShallow({String joiner: ',
', DiagnosticLevel minLevel: DiagnosticLevel.debug }) → String
Returns a one-line detailed description of the object. [...]

inherited

toStringShort() → String
A short, textual description of this widget.

inherited

Operators

operator ==(dynamic other) → bool
The equality operator. [...]

inherited

Cours de sémantique

Un widget qui annote l’arborescence des widgets avec une description de la signification des widgets.

Utilisé par les outils d'accessibilité, les moteurs de recherche et d'autres logiciels d'analyse sémantique pour déterminer la signification de l'application.

Voir également:

MergeSemantics, qui marque un sous-arbre comme étant un seul nœud à des fins d'accessibilité.

ExcludeSemantics, qui exclut un sous-arbre de l’arborescence sémantique (ce qui peut être utile s’il est, par exemple, totalement décoratif et sans importance pour l’utilisateur).

RenderObject.semanticsAnnotator, l'API de bibliothèque de rendu à travers laquelle le widget Sémantique est réellement implémenté.

SemanticsNode, l'objet utilisé par la bibliothèque de rendu pour représenter la sémantique dans l'arbre sémantique.

SemanticsDebugger, une superposition permettant de visualiser l’arborescence de la sémantique. Peut être activé à l'aide de WidgetsApp.showSemanticsDebugger ou MaterialApp.showSemanticsDebugger.

Héritage

Objet Diagnosticable DiagnosticableTree Widget RenderObjectWidget SingleChildRenderObjectWidget Sémantique

Annotations

@immutable

Constructeurs

Sémantique ({Key key, Widget child, bool container: false, bool explicitChildNodes: false, bool excludeSemantics: false, bool activé, bool coché, bool coché, bool basculé, bool bouton, bool en-tête, bool textField, bool readOnly, bool focus, , bool inMutuallyExclusiveGroup, bool obscured, bool scopesRoute, bool namesRoute, bool hidden, bool image, bool liveRegion, étiquette de chaîne, valeur de chaîne, String onTap, VoidCallback onLongPress, VoidCallback onScrollLeft, VoidCallback onScrollRight, VoidCallback onScrollUp, VoidCallback onScrollDown, VoidCallback onIncrease, VoidCallback onDecrease, VoidCallback oncopy, VoidCallback onCut, VoidCallback onpaste, VoidCallback onDismiss, MoveCursorHandler onMoveCursorForwardByCharacter, MoveCursorHandler onMoveCursorBackwardByCharacter, SetSelectionHandler onSetSelection, VoidCallback onDidG ainAccessibilityFocus, VoidCallback onDidLoseAccessibilityFocus, Map <CustomSemanticsAction, VoidCallback> customSemanticsActions})

Crée une annotation sémantique. [...]

Semantics.fromProperties ({Key, enfant Widget, conteneur bool: false, bool explicitChildNodes: false, bool excludeSemantics: false, @ propriétés SemanticsProperties requises})

Crée une annotation sémantique à l'aide de SemanticsProperties. [...]

const

Propriétés

conteneur → bool

Si container est true, ce widget introduit un nouveau nœud dans l'arborescence sémantique. Sinon, la sémantique sera fusionnée avec la sémantique de tous les ancêtres (si l'ancêtre le permet). [...]

final

excludeSemantics → bool

S'il faut remplacer toutes les sémantiques enfants par ce nœud. [...]

final

explicitChildNodes → bool

Indique si les descendants de ce widget sont autorisés à ajouter des informations sémantiques au SemanticsNode annoté par ce widget. [...]

final

propriétés → SemanticsProperties

Contient les propriétés utilisées par les technologies d'assistance pour rendre l'application plus accessible.

final

enfant → widget

Le widget situé sous ce widget dans l'arborescence. [...]

final, hérité

hashCode → int

Le code de hachage pour cet objet. [...]

en lecture seule, hérité

clé → clé

Contrôle la manière dont un widget remplace un autre widget dans l'arborescence. [...]

final, hérité

runtimeType → Type

Une représentation du type d'exécution de l'objet.

en lecture seule, hérité

Les méthodes

createRenderObject (contexte BuildContext) → RenderSemanticsAnnotations

Crée une instance de la classe RenderObject que ce RenderObjectWidget représente, à l'aide de la configuration décrite par ce RenderObjectWidget. [...]

passer outre

debugFillProperties (propriétés de DiagnosticPropertiesBuilder) → void

Ajoutez des propriétés supplémentaires associées au nœud. [...]

passer outre

updateRenderObject (contexte BuildContext, covariant RenderSemanticsAnnotations renderObject) → void

Copie la configuration décrite par ce RenderObjectWidget dans le RenderObject donné, qui sera du même type que celui renvoyé par createRenderObject de cet objet. [...]

passer outre

createElement () → SingleChildRenderObjectElement

RenderObjectWidgets se gonfle toujours dans une sous-classe RenderObjectElement.

hérité

debugDescribeChildren () → Liste <DiagnosticsNode>

Renvoie une liste d'objets DiagnosticsNode décrivant les enfants de ce nœud. [...]

@protected, hérité

didUnmountRenderObject (RenderObject covariant renderObject) → void

Un objet de rendu précédemment associé à ce widget a été supprimé de l'arborescence. Le RenderObject donné sera du même type que celui renvoyé par createRenderObject de cet objet.

@protected, hérité

noSuchMethod (invocation d'invocation) → dynamique

Appelé lors de l'accès à une méthode ou à une propriété non existante. [...]

hérité

toDiagnosticsNode ({Nom de la chaîne, style DiagnosticsTreeStyle}) → DiagnosticsNode

Renvoie une représentation de débogage de l'objet utilisée par les outils de débogage et par DiagnosticsNode.toStringDeep. [...]

hérité

toString ({DiagnosticLevel minLevel: DiagnosticLevel.debug})

Animation and motion widgets

1. Documentation
2. Développement
3. Interface utilisateur
4. Widgets
5. Animation

Bring animations to your app.

See more widgets in the widget catalog.

AnimatedBuilder

A general-purpose widget for building animations. AnimatedBuilder is useful for more complex widgets that wish to include an animation as part of a larger build function. To use AnimatedBuilder, simply construct the widget and pass it a builder function.

AnimatedBuilder

Un widget à usage général pour la création d'animations. AnimatedBuilder est utile pour les widgets plus complexes qui souhaitent inclure une animation dans le cadre d'une fonction de construction plus grande. Pour utiliser AnimatedBuilder, construisez simplement le widget et transmettez-lui une fonction de générateur.

AnimatedContainer

A container that gradually changes its values over a period of time.

Conteneur Animé

Un conteneur qui modifie progressivement ses valeurs sur une période donnée.

AnimatedCrossFade

A widget that cross-fades between two given children and animates itself between their sizes.

AnimatedCrossFade

Un widget qui effectue un fondu enchaîné entre deux enfants donnés et s'anime entre leurs tailles.

AnimatedDefaultTextStyle

Animated version of DefaultTextStyle which automatically transitions the default text style (the text style to apply to descendant Text widgets without explicit style) over a given duration whenever the given style changes.

AnimatedDefaultTextStyle

Version animée de DefaultTextStyle qui transforme automatiquement le style de texte par défaut (le style de texte à appliquer aux widgets de texte descendants sans style explicite) sur une durée donnée, chaque fois que le style donné est modifié.

AnimatedListState

The state for a scrolling container that animates items when they are inserted or removed.

AnimatedListState

Etat d'un conteneur de défilement qui anime les éléments lorsqu'ils sont insérés ou supprimés.

AnimatedModalBarrier

A widget that prevents the user from interacting with widgets behind itself.

AnimatedModalBarrier

Un widget qui empêche l'utilisateur d'interagir avec les widgets derrière lui.

AnimatedOpacity

Animated version of Opacity which automatically transitions the child's opacity over a given duration whenever the given opacity changes.

AnimatedOpacity

Version animée de l'opacité qui fait automatiquement passer l'opacité de l'enfant sur une durée donnée, chaque fois que l'opacité donnée change.

AnimatedPhysicalModel

Animated version of PhysicalModel.

AnimatedPhysicalModel

Version animée de PhysicalModel.

AnimatedPositioned

Animated version of Positioned which automatically transitions the child's position over a given duration whenever the given position changes.

AniméPositionné

Version animée de Positionné qui transforme automatiquement la position de l'enfant sur une durée donnée à chaque changement de position.

AnimatedSize

Animated widget that automatically transitions its size over a given duration whenever the given child's size changes.

Taille animée

Widget animé qui transforme automatiquement sa taille sur une durée donnée à chaque changement de taille de l'enfant.

AnimatedWidget

A widget that rebuilds when the given Listenable changes value.

Voyage animé

Un widget qui se reconstruit lorsque le paramètre Listenable donné change de valeur.

AnimatedWidgetBaseState

A base class for widgets with implicit animations.

AnimatedWidgetBaseState

Une classe de base pour les widgets avec des animations implicites.

DecoratedBoxTransition

Animated version of a DecoratedBox that animates the different properties of its Decoration.

DécoréBoxTransition

Version animée d'une DecoratedBox qui anime les différentes propriétés de sa décoration.

FadeTransition

Animates the opacity of a widget.

FadeTransition

Anime l'opacité d'un widget.

Hero

A widget that marks its child as being a candidate for hero animations.

héros

Un widget qui marque son enfant en tant que candidat pour les animations de héros.

PositionedTransition

Animated version of Positioned which takes a specific Animation to transition the child's position from a start position to and end position over the lifetime of the animation.

PositionedTransition

Version animée de Positionné qui prend une animation spécifique pour faire passer la position de l'enfant d'une position de départ à une position de fin sur la durée de vie de l'animation.

RotationTransition

Animates the rotation of a widget.

RotationTransition

Anime la rotation d'un widget.

ScaleTransition

Animates the scale of transformed widget.

ScaleTransition

Anime l'échelle du widget transformé.

SizeTransition

Animates its own size and clips and aligns the child.

TailleTransition

Anime sa propre taille et clips et aligne l'enfant.

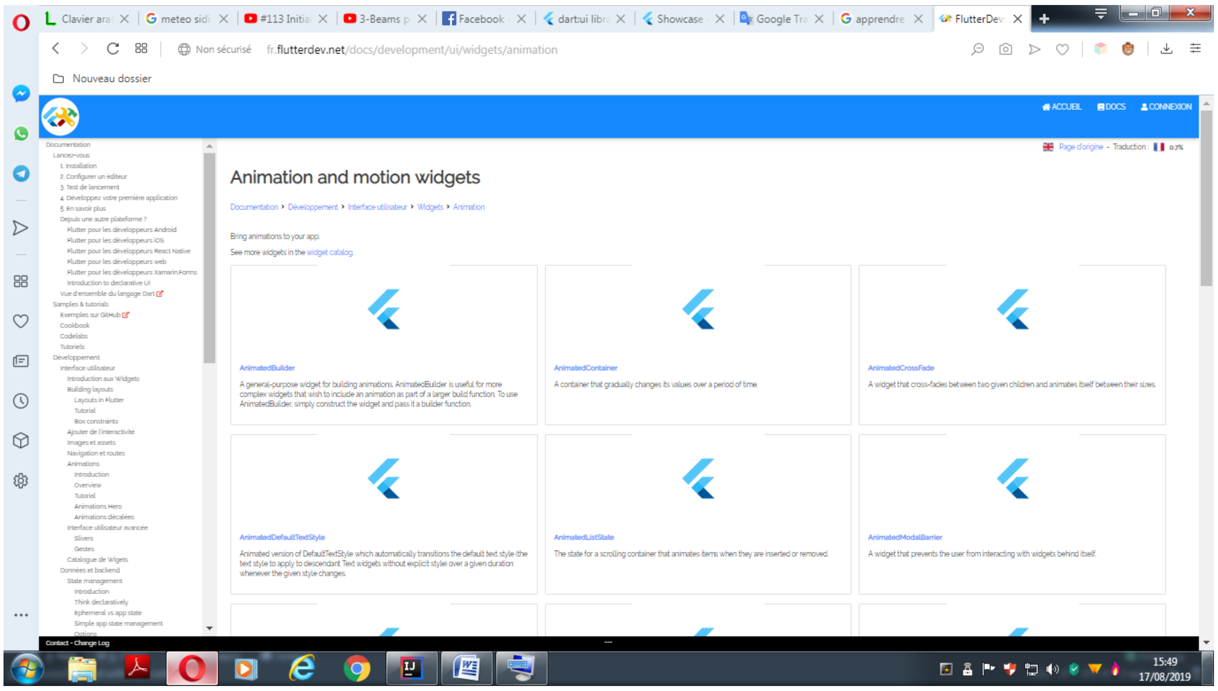
SlideTransition

Animates the position of a widget relative to its normal position.

Widgets d'animation et de mouvement

Transition de diapositive

Anime la position d'un widget par rapport à sa position normale


fr.flutterdev.net/docs/development/ui/widgets/animation

Animation and motion widgets

Documentation › Développement › Interface utilisateur › Widgets › Animation

Bring animations to your app.

See more widgets in the widget catalog.

AnimatedBuilder
A general-purpose widget for building animations. AnimatedBuilder is useful for more complex widgets that wish to include an animation as part of a larger build function. To use AnimatedBuilder, simply construct the widget and pass it a builder function.

AnimatedContainer
A container that gradually changes its values over a period of time.

AnimatedCrossFade
A widget that cross-fades between two given children and animates itself between their sizes.

AnimatedDefaultTextStyle
Animated version of DefaultTextStyle which automatically transitions the default text style (the text style to apply to descendant Text widgets without explicit style) over a given duration whenever the given style changes.

AnimatedListState
The state for a scrolling container that animates items when they are inserted or removed.

AnimatedModalBarrier
A widget that prevents the user from interacting with widgets behind itself.

AnimatedBuilder class

A general-purpose widget for building animations.

AnimatedBuilder is useful for more complex widgets that wish to include an animation as part of a larger build function. To use AnimatedBuilder, simply construct the widget and pass it a builder function.

For simple cases without additional state, consider using AnimatedWidget.

Performance optimizations

If your builder function contains a subtree that does not depend on the animation, it's more efficient to build that subtree once instead of rebuilding it on every animation tick.

If you pass the pre-built subtree as the child parameter, the AnimatedBuilder will pass it back to your builder function so that you can incorporate it into your build.

Using this pre-built child is entirely optional, but can improve performance significantly in some cases and is therefore a good practice.

Sample

This code defines a widget called `Spinner` that spins a green square continually. It is built with an AnimatedBuilder and makes use of the child feature to avoid having to rebuild the Container each time.

```
class Spinner extends StatefulWidget {
  @override
  _SpinnerState createState() => _SpinnerState();
}

class _SpinnerState extends State<Spinner> with
SingleTickerProviderStateMixin {
  AnimationController _controller;

  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      duration: const Duration(seconds: 10),
      vsync: this,
    )..repeat();
  }

  @override
  void dispose() {
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return AnimatedBuilder(
      animation: _controller,
      child: Container(width: 200.0, height: 200.0, color:
Colors.green),
      builder: (BuildContext context, Widget child) {
        return Transform.rotate(
          angle: _controller.value * 2.0 * math.pi,
          child: child,
        );
      },
    );
  }
}
```

Classe AnimatedBuilder

Un widget à usage général pour la création d'animations.

AnimatedBuilder est utile pour les widgets plus complexes qui souhaitent inclure une animation dans le cadre d'une fonction de construction plus grande. Pour utiliser AnimatedBuilder, construisez simplement le widget et transmettez-lui une fonction de générateur.

Pour les cas simples sans état supplémentaire, envisagez d'utiliser AnimagedWidget.

Optimisations de performances

Si votre fonction de générateur contient une sous-arborescence qui ne dépend pas de l'animation, il est plus efficace de construire cette sous-arborescence une fois au lieu de la reconstruire à chaque tick de l'animation.

Si vous transmettez le sous-arbre prédéfini en tant que paramètre enfant, AnimatedBuilder le transmettra à votre fonction de générateur afin que vous puissiez l'intégrer à votre construction.

L'utilisation de cet enfant prédéfini est entièrement facultative, mais peut dans certains cas améliorer considérablement les performances et constitue donc une bonne pratique.

Échantillon

Ce code définit un widget appelé Spinner qui tourne continuellement un carré vert. Il est construit avec un AnimatedBuilder et utilise la fonctionnalité enfant pour éviter de devoir reconstruire le conteneur à chaque fois.

affectation

```
La classe Spinner étend StatefulWidget {

  @passer outre

  _SpinnerState createState () => _SpinnerState ();

}
```

Assets, images, and icon widgets

1. Documentation
2. Développement
3. Interface utilisateur
4. Widgets
5. Assets

Manage assets, display images, and show icons.

See more widgets in the widget catalog.

AssetBundle

Asset bundles contain resources, such as images and strings, that can be used by an application. Access to these resources is asynchronous so that they can be transparently loaded over a network (e.g., from a NetworkAssetBundle) or from the local file system without blocking the application's user interface.

Icon

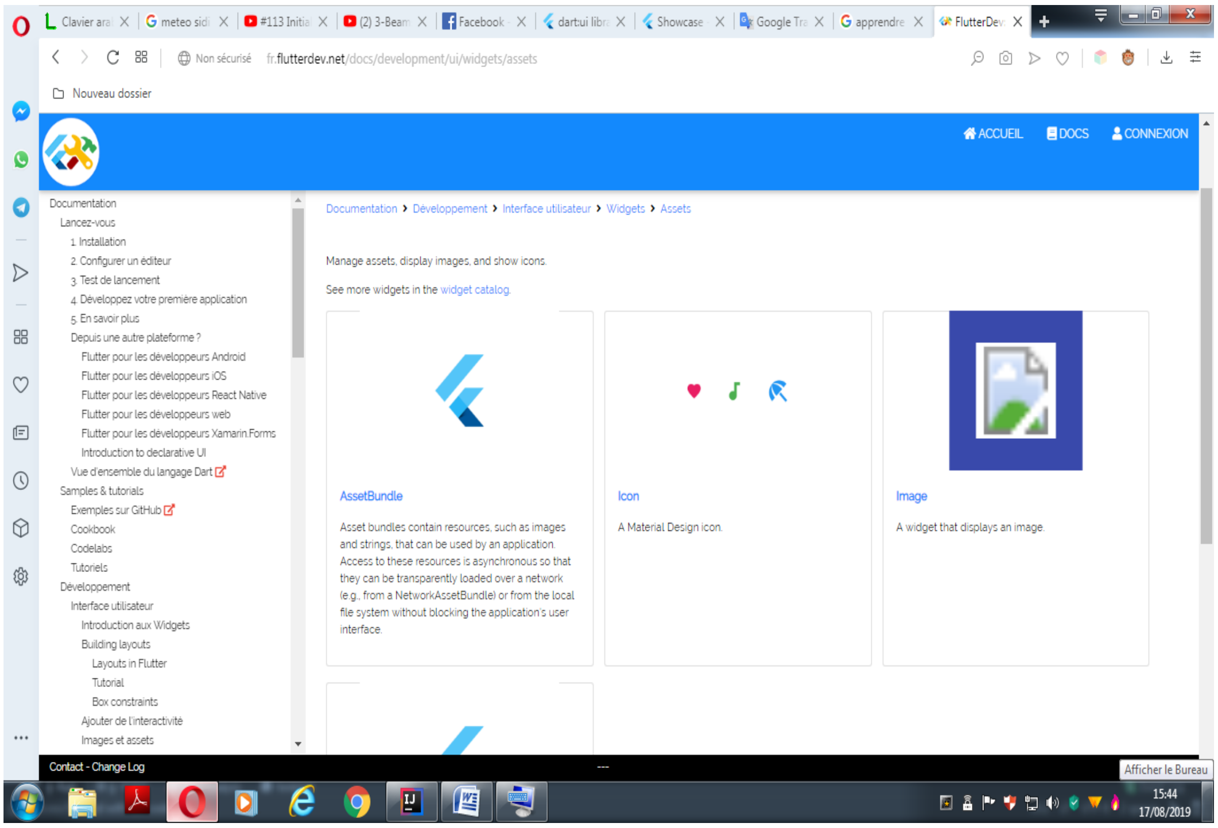
A Material Design icon.

Image

A widget that displays an image.

RawImage

A widget that displays a dart:ui.Image directly.


fr.flutterdev.net/docs/development/ui/widgets/assets
ACCUEIL
DOCS
CONNEXION
Documentation
Lancez-vous
1. Installation
2. Configurer un éditeur
3. Test de lancement
4. Développez votre première application
5. En savoir plus
Depuis une autre plateforme ?
Flutter pour les développeurs Android
Flutter pour les développeurs iOS
Flutter pour les développeurs React Native
Flutter pour les développeurs web
Flutter pour les développeurs Xamarin.Forms
Introduction to declarative UI
Vue d'ensemble du langage Dart
Samples & tutorials
Exemples sur GitHub
Cookbook
Codelabs
Tutoriels
Développement
Interface utilisateur
Introduction aux Widgets
Building layouts
Layouts in Flutter
Tutorial
Box constraints
Ajouter de l'interactivité
Images et assets
Contact - Change Log
Documentation › Développement › Interface utilisateur › Widgets › Assets

Manage assets, display images, and show icons.

See more widgets in the widget catalog.

AssetBundle

Asset bundles contain resources, such as images and strings, that can be used by an application. Access to these resources is asynchronous so that they can be transparently loaded over a network (e.g., from a NetworkAssetBundle) or from the local file system without blocking the application's user interface.

Icon

A Material Design icon.

Image

A widget that displays an image.

Gérez les ressources, affichez les images et affichez les icônes.

Voir plus de widgets dans le catalogue de widgets.

AssetBundle

Les ensembles d'actifs contiennent des ressources, telles que des images et des chaînes, pouvant être utilisées par une application. L'accès à ces ressources est asynchrone, de sorte qu'elles peuvent être chargées de manière transparente sur un réseau (par exemple, à partir d'un NetworkAssetBundle) ou à partir du système de fichiers local sans bloquer l'interface utilisateur de l'application.

Icône

Une icône de conception matérielle.

Image

Un widget qui affiche une image.

.

Async

Motifs asynchrones dans votre application Flutter.

Async widgets

1. Documentation
2. Développement
3. Interface utilisateur
4. Widgets
5. Async

Async patterns to your Flutter application.

See more widgets in the widget catalog.

FutureBuilder

Widget that builds itself based on the latest snapshot of interaction with a Future.

StreamBuilder

Widget that builds itself based on the latest snapshot of interaction with a Stream.

See more widgets in the widget catalog.

Widgets Async

Documentation

Développement

Interface utilisateur

Widgets

Async

Motifs asynchrones dans votre application Flutter.

Voir plus de widgets dans le catalogue de widgets.

FutureBuilder

Widget qui se construit à partir du dernier instantané d'interaction avec un futur.

StreamBuilder

Widget qui se construit à partir du dernier instantané d'interaction avec un flux.

Voir plus de widgets dans le catalogue de widgets.

Widgets de base

Les widgets que vous devez absolument connaître avant de créer votre première application Flutter.

Cupertino (widget iOS-style)

Des widgets magnifiques et haute fidélité pour être au plus proche du desing iOS actuel.

Input

Saisissez les entrées de l'utilisateur en plus des widgets d'entrée Material et Cupertino.

Modèles d'interaction

Répondre aux événements tactiles et orientez les utilisateurs vers différentes vues.

Layout

Placer les autres widgets, en colonnes, en lignes, en grilles et autres dispositions.

Composants Material

Widgets visuels, comportementaux et riches en mouvements implémentant les directives de conception Material.

Peinture et effets

Ces widgets appliquent des effets visuels aux widgets enfants sans modifier leur disposition, leur taille ou leur position.

Scrolling

Faites défiler plusieurs widgets en tant qu'enfants du parent.

Styliser

Gérez le thème de votre application, adaptez votre application aux tailles d'écran ou ajoutez un marge interne.

Texte

Afficher et styliser le texte.