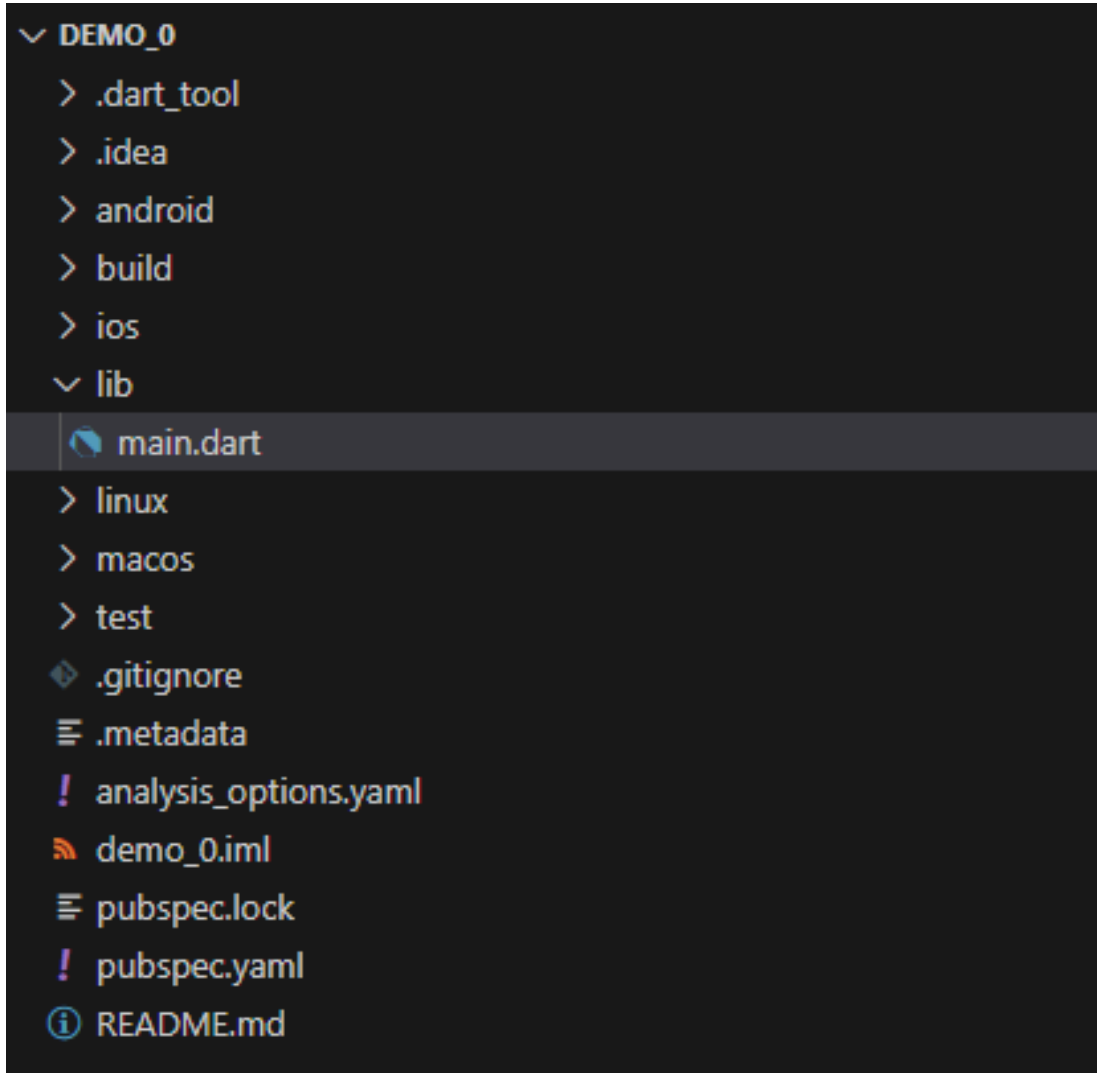
Chapitre 5 : Développement Flutter

Plan

1. Structure d'un projet flutter
2. Les widgets
3. Gestion des listes
4. Navigation
5. Création des formulaires
6. Les animations et les transitions
7. Gestion avancée de l'état avec provider

Les bases de Flutter

Structure d'un projet Flutter



Structure d'un projet Flutter

Présentation des dossiers principaux :

- lib/ : Contient le code source de l'application. C'est ici que les fichiers Dart seront principalement créés.
- pubspec.yaml : Le fichier de configuration du projet où vous gérez les dépendances, les assets, etc.
- android/ et ios/ : Contiennent les configurations spécifiques aux plateformes Android et iOS.
- test/ : Dossier pour les tests unitaires et widgets.
- main.dart : Le point d'entrée de l'application.
- build.gradle, AndroidManifest.xml, Info.plist, etc., pour les configurations spécifiques aux plateformes.

Introduction aux Widgets

- Dans Flutter, les widgets sont les éléments de base de l'interface utilisateur. Un texte, une image, un bouton, ou même une mise en page, est représenté par un widget.

Comprendre les widgets est essentiel pour maîtriser le développement d'applications Flutter

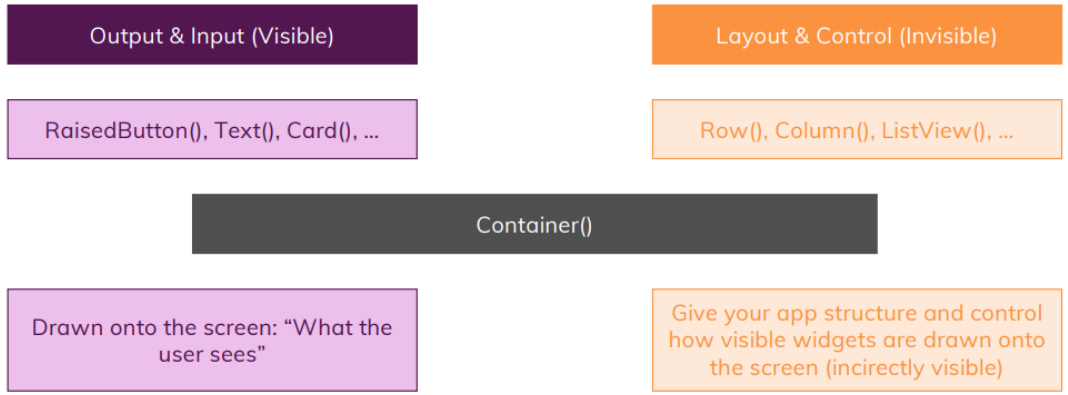
Qu'est-ce qu'un Widget ?

- Un widget est une classe dans Flutter qui décrit une partie de l'interface utilisateur. Les widgets sont organisés en un arbre hiérarchique, souvent appelé "arbre des widgets" (widget tree). Chaque widget dans cet arbre est responsable de dessiner une partie spécifique de l'interface utilisateur.

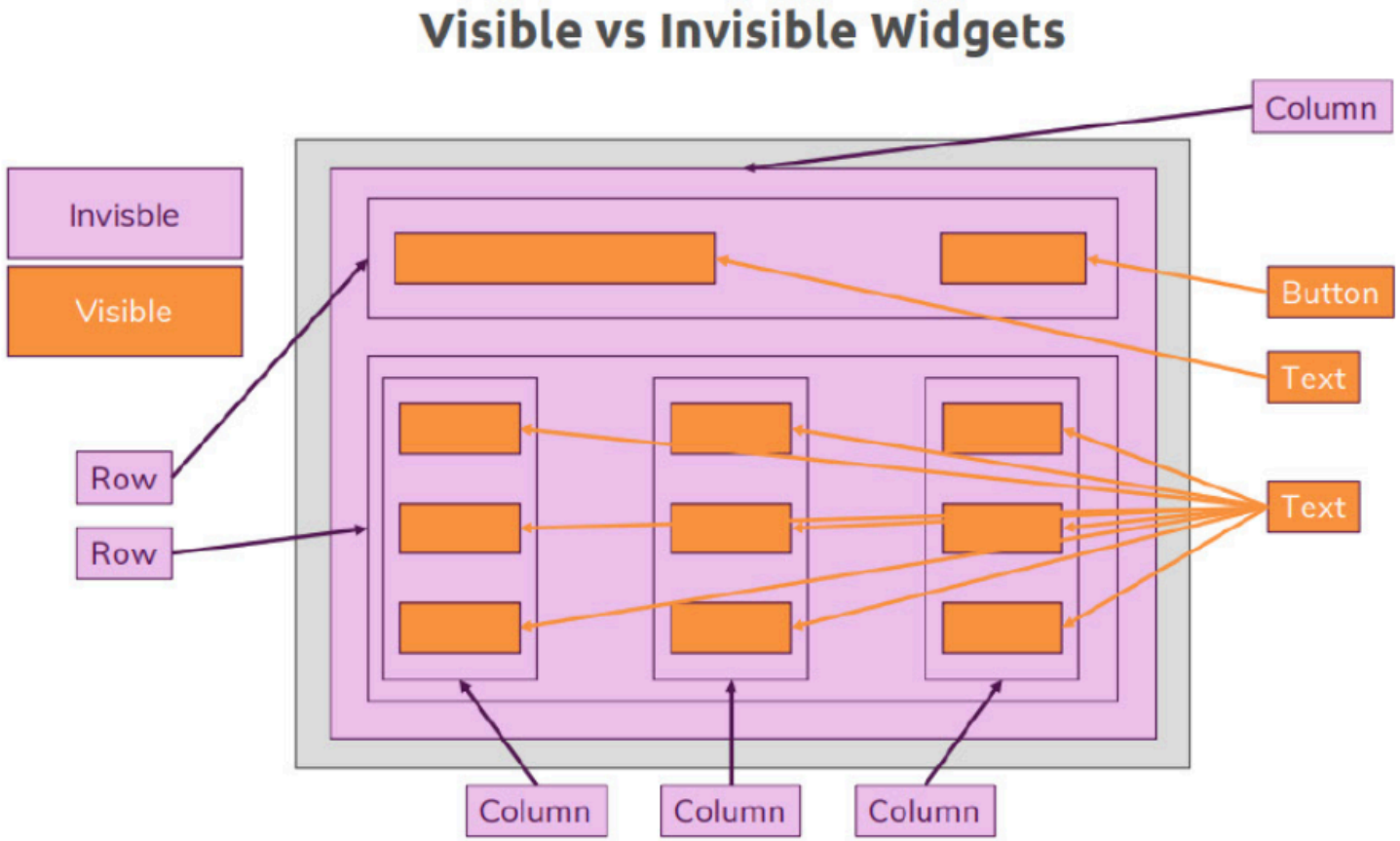
Introduction aux Widgets

Les widgets peuvent être :

Visuels : Comme les boutons, les textes, les images, etc.
Structuraux : Comme les colonnes (Column), les lignes (Row), les conteneurs (Container), qui définissent la disposition des widgets enfants.



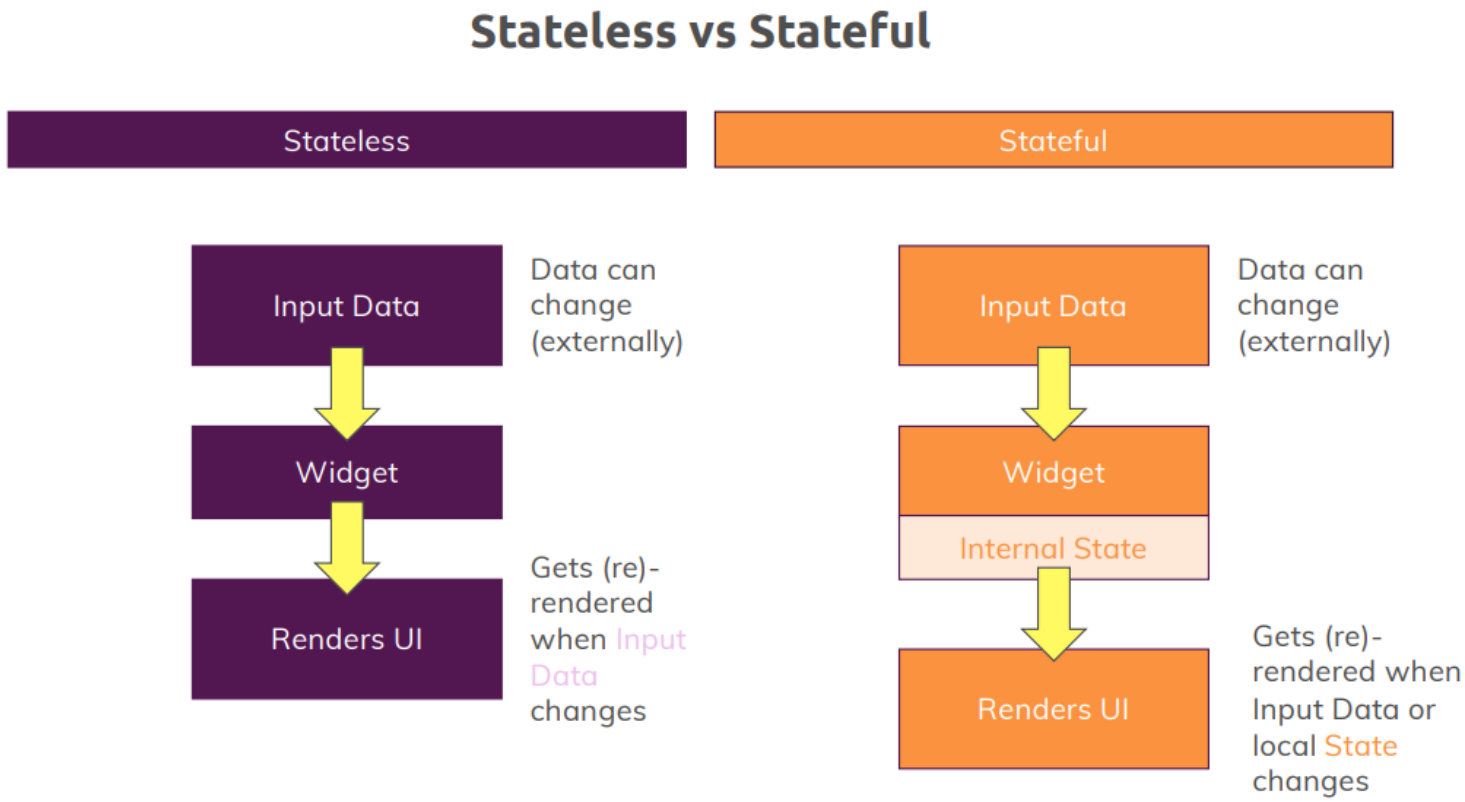
Introduction aux Widgets



Introduction aux Widgets

Les différents types de Widgets

- StatelessWidget (Widgets sans état)
- StatefulWidget (Widgets avec état)



Introduction aux Widgets

StatelessWidget : Les Widgets sans état

- Un StatelessWidget est un widget qui ne peut pas changer après avoir été initialisé.
- Un widget sans état ne réagit pas aux changements dynamiques pendant l'exécution de l'application. => utilisés lorsque l'interface utilisateur reste constante.

Dans cet exemple, le texte "Hello, World!" est affiché au centre de l'écran, et il ne changera jamais car MonWidgetStateless est un StatelessWidget.

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

class MonWidgetStateless extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        'Hello, World!',
        style: TextStyle(fontSize: 24),
      ),
    );
  }
}
```

Introduction aux Widgets

StatefulWidget : Les Widgets avec état

- Un StatefulWidget est un widget qui peut changer son état au cours du temps, par exemple, lorsqu'un utilisateur interagit avec lui. Chaque StatefulWidget est associé à un objet State, qui gère les changements d'état et reconstruit l'interface utilisateur chaque fois que l'état change.

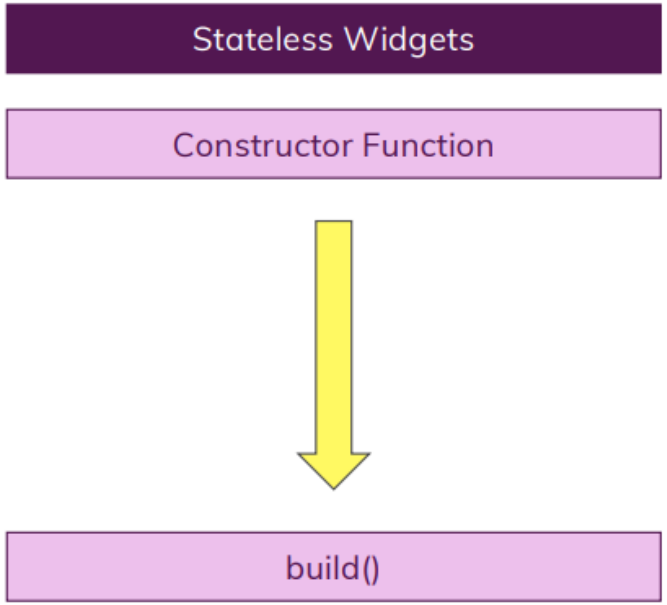
```
import 'package:flutter/material.dart';

class MonWidgetStateful extends StatefulWidget {
  @override
  _MonWidgetStatefulState createState() => _MonWidgetStatefulState();
}

class _MonWidgetStatefulState extends State<MonWidgetStateful> {
  int _compteur = 0;

  void _incrementerCompteur() {
    setState(() {
      _compteur++;
    });
  }
```

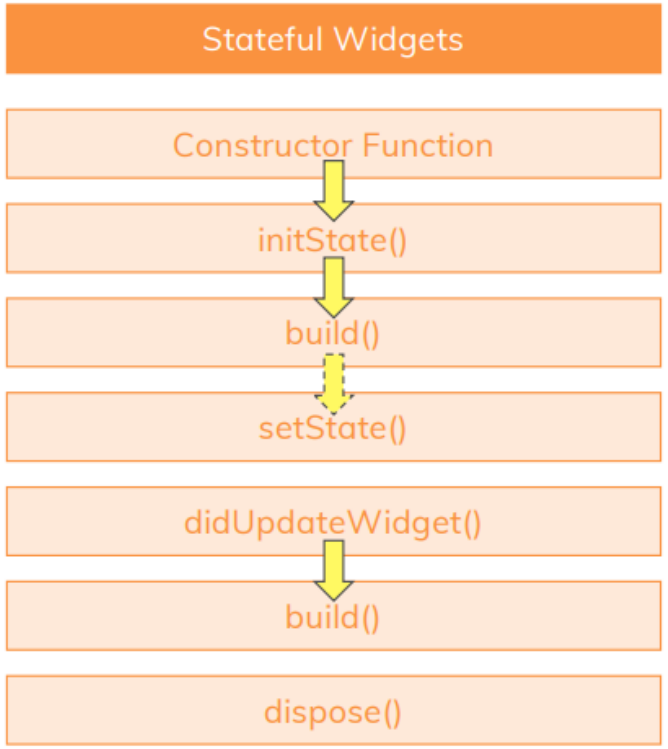
Introduction aux Widgets

Le Cycle de Vie d'un StatefulWidget

- createState() : Lorsque le widget est créé, la méthode createState() est appelée pour créer une instance de l'objet State.
- initState() : La méthode initState() est appelée une seule fois, lors de la création de l'état. C'est ici que vous pouvez initialiser des données.
- build() : La méthode build() est appelée chaque fois que l'interface utilisateur doit être reconstruite, par exemple après un appel à setState().
- dispose() : La méthode dispose() est appelée lorsque le widget est retiré de l'arbre des widgets, généralement pour libérer les ressources utilisées par le widget.

Introduction aux Widgets





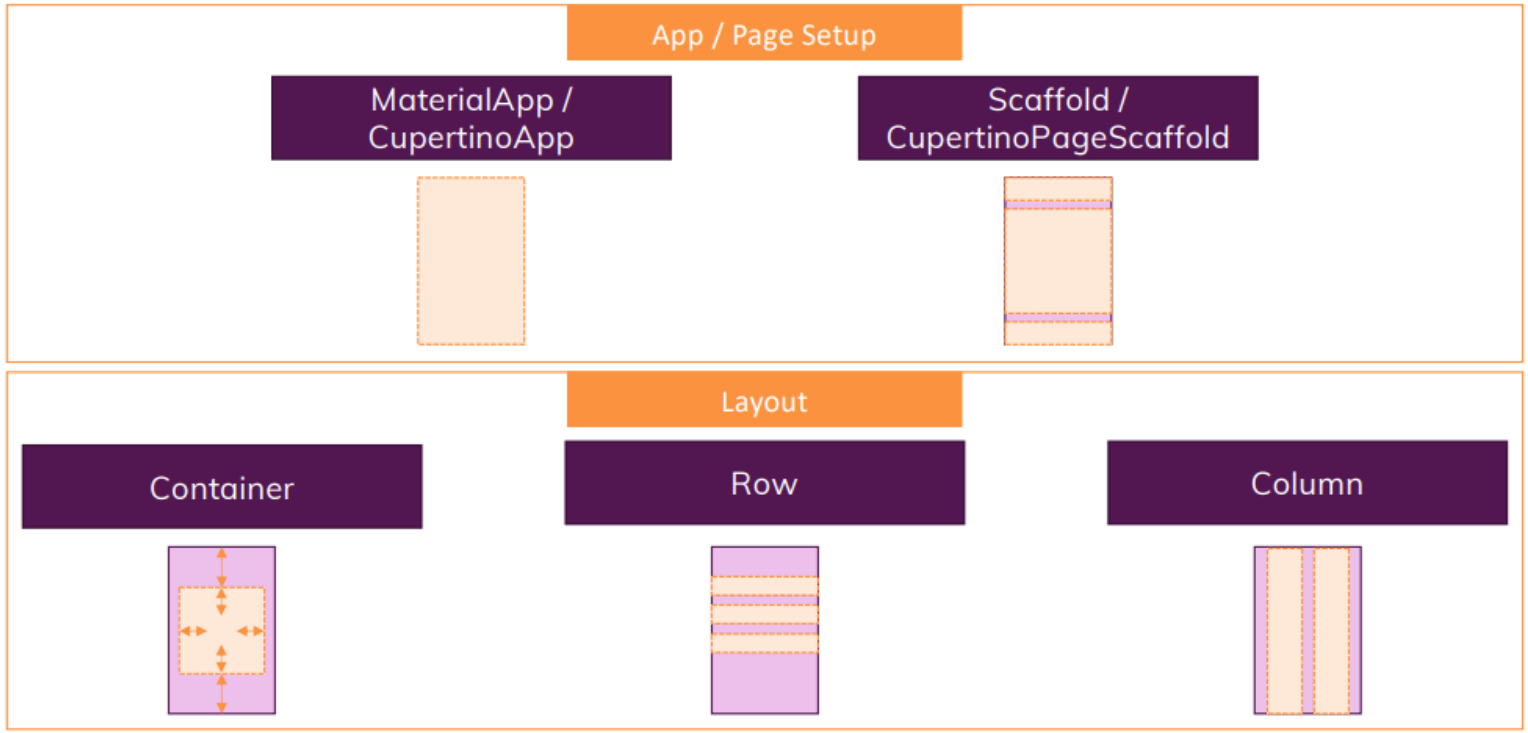
Introduction aux Widgets

Pourquoi les Widgets sont-ils si importants ?

- Modularité et réactivité des UI: Chaque composant de l'interface peut être isolé et réutilisé, ce qui rend le code plus maintenable et plus facile à comprendre.
- Composition : Les widgets peuvent être composés les uns avec les autres pour créer des interfaces complexes.
- Personnalisation : Grâce aux nombreuses options de personnalisation, les widgets peuvent être stylisés pour correspondre à n'importe quelle charte graphique.
- Portabilité : Comme les widgets contrôlent chaque pixel à l'écran, ils permettent de créer des interfaces utilisateur cohérentes sur toutes les plateformes (Android, iOS, Web, Desktop).

Most Important Widgets

Introduction aux Widgets



Introduction aux Widgets

MaterialApp / CupertinoApp

- Généralement le widget racine de votre application
- Fait beaucoup de travail de configuration "en coulisses" pour votre application => permet de configurer un thème global pour votre application
- Configure le comportement de la navigation (par exemple, les animations) pour votre application

Scaffold / CupertinoPageScaffold

- Généralement utilisé comme cadre pour une page dans votre application
- Fournit un arrière-plan, une barre d'application, des onglets de navigation, etc.
- N'utilisez qu'un seul scaffold par page !

Introduction aux Widgets

Container

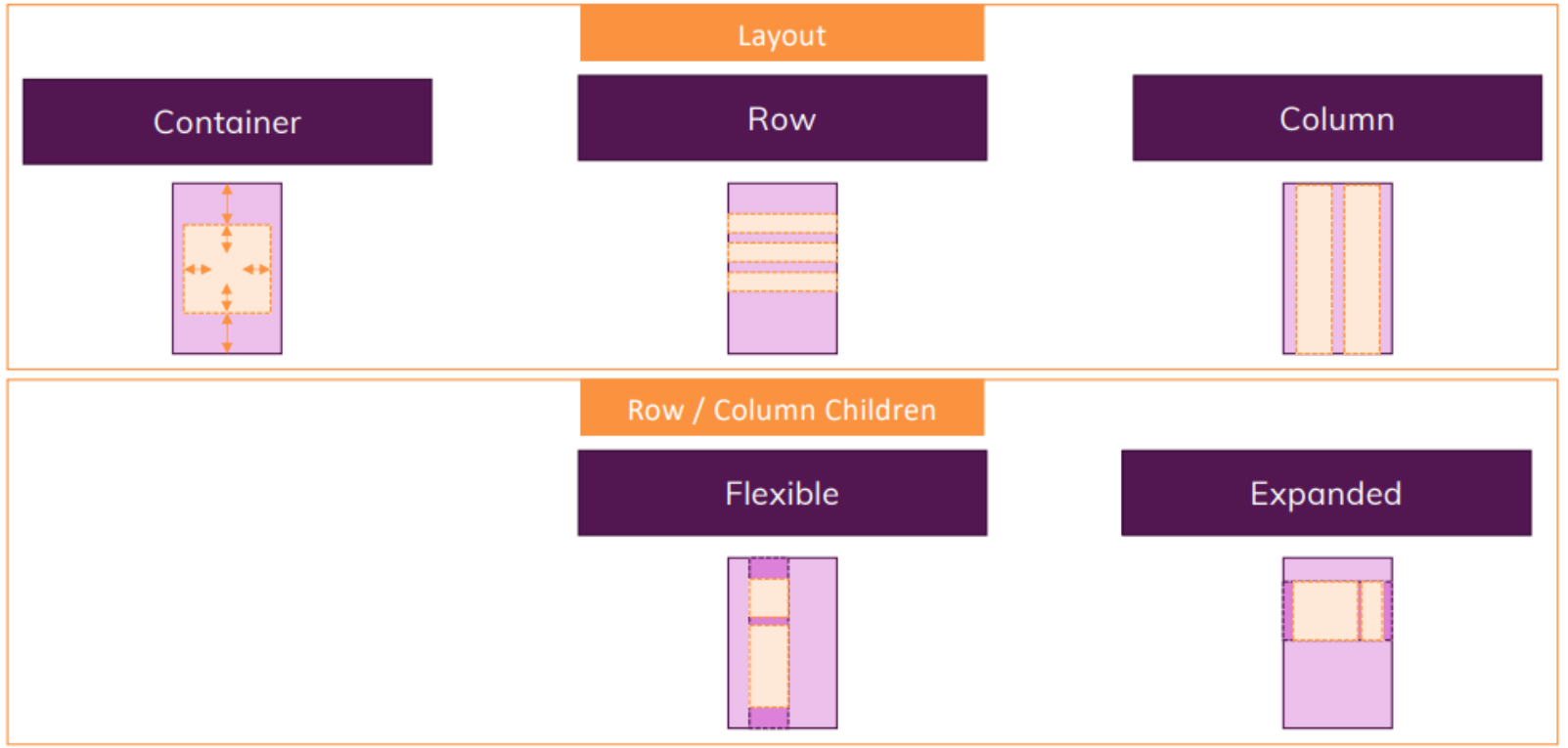
- Widget extrêmement polyvalent !
- Peut être dimensionné (largeur, hauteur, maxWidth, maxHeight), stylisé (bordure, couleur, forme, ...) et plus encore
- Peut prendre un enfant (mais n'est pas obligé), que vous pouvez également aligner de différentes manières

Row / Column

- À utiliser si vous avez besoin que plusieurs widgets soient placés côte à côte horizontalement ou verticalement
- Options de style limitées => Enveloppez avec un Container (ou enveloppez les widgets enfants) pour appliquer du style
- Les enfants peuvent être alignés le long de l'axe principal et de l'axe croisé

Most Important Widgets

Introduction aux Widgets



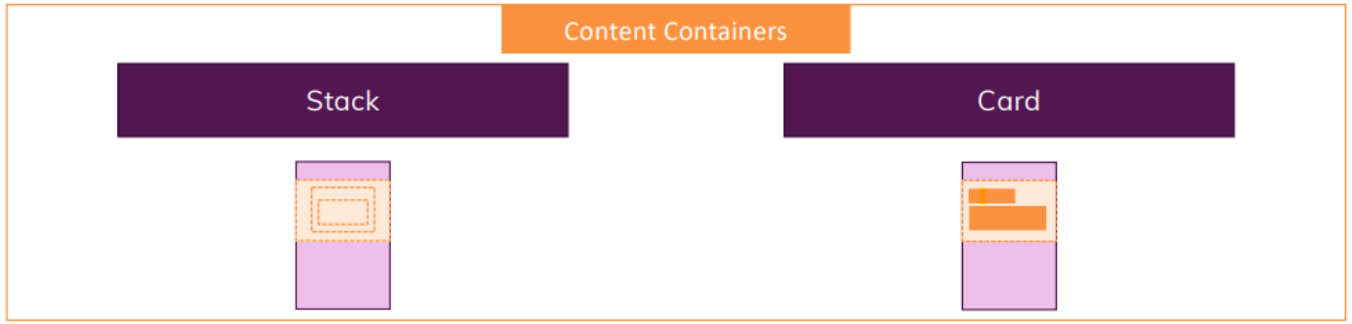
Introduction aux Widgets

Flexible / Expanded

- Expanded : Utile pour remplir une zone (force un enfant à occuper tout un espace)
- Flexible: le widget s'adapte mais reste flexible

Most Important Widgets

Introduction aux Widgets



Introduction aux Widgets

Stack

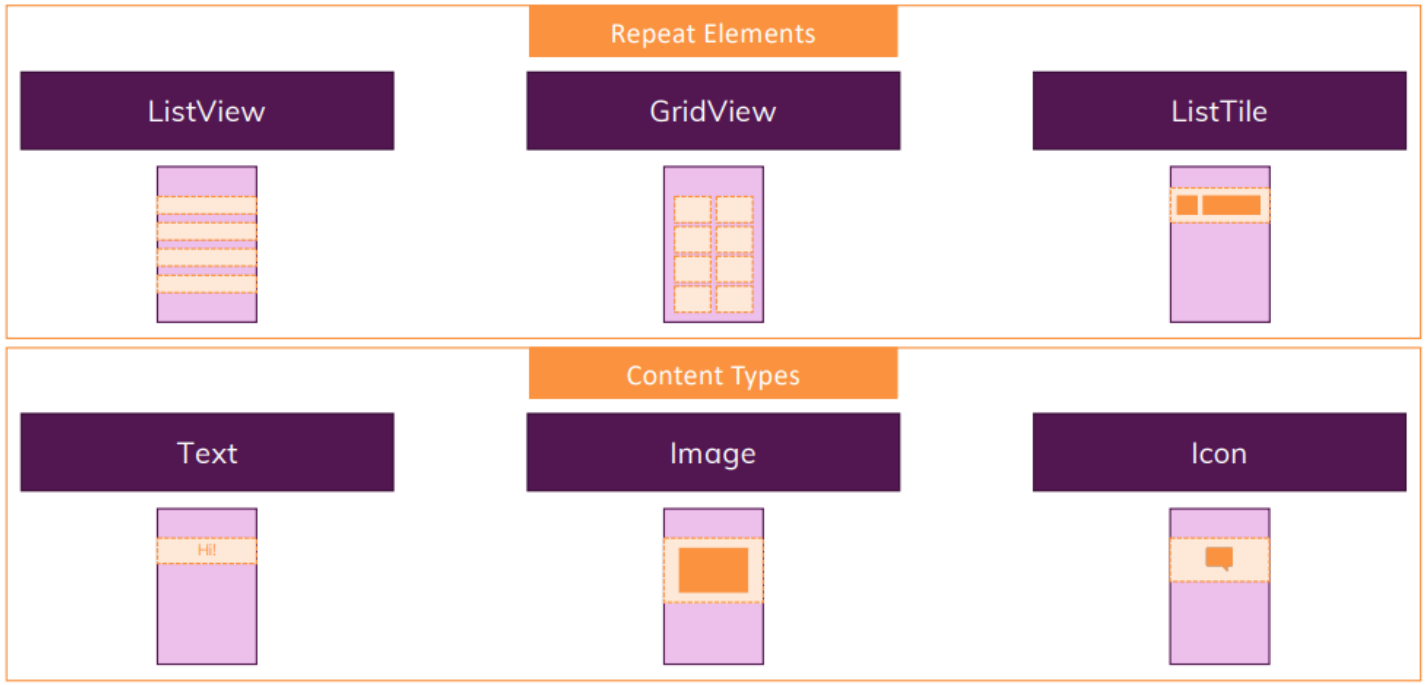
- Utilisé pour positionner des éléments les uns sur les autres (le long de l'axe Z) : Les widgets peuvent se chevaucher
- Vous pouvez positionner des éléments dans un espace absolu (c'est-à-dire dans un espace de coordonnées) via le widget Positioned()

Card

- Un conteneur avec un certain style par défaut (ombre, couleur de fond, coins arrondis)
- Peut contenir un enfant (sans restriction)
- Typiquement utilisé pour afficher un seul élément / groupe d'informations

Les Widgets les Plus Importants dans Flutter

Introduction aux Widgets



Introduction aux Widgets

Text

- Un widget qui affiche simplement du texte à l'écran
- Le texte peut être stylisé (famille de polices, épaisseur de la police, taille de la police, etc.). Le comportement du texte peut être contrôlé (par exemple, découpage si c'est trop long)

Image

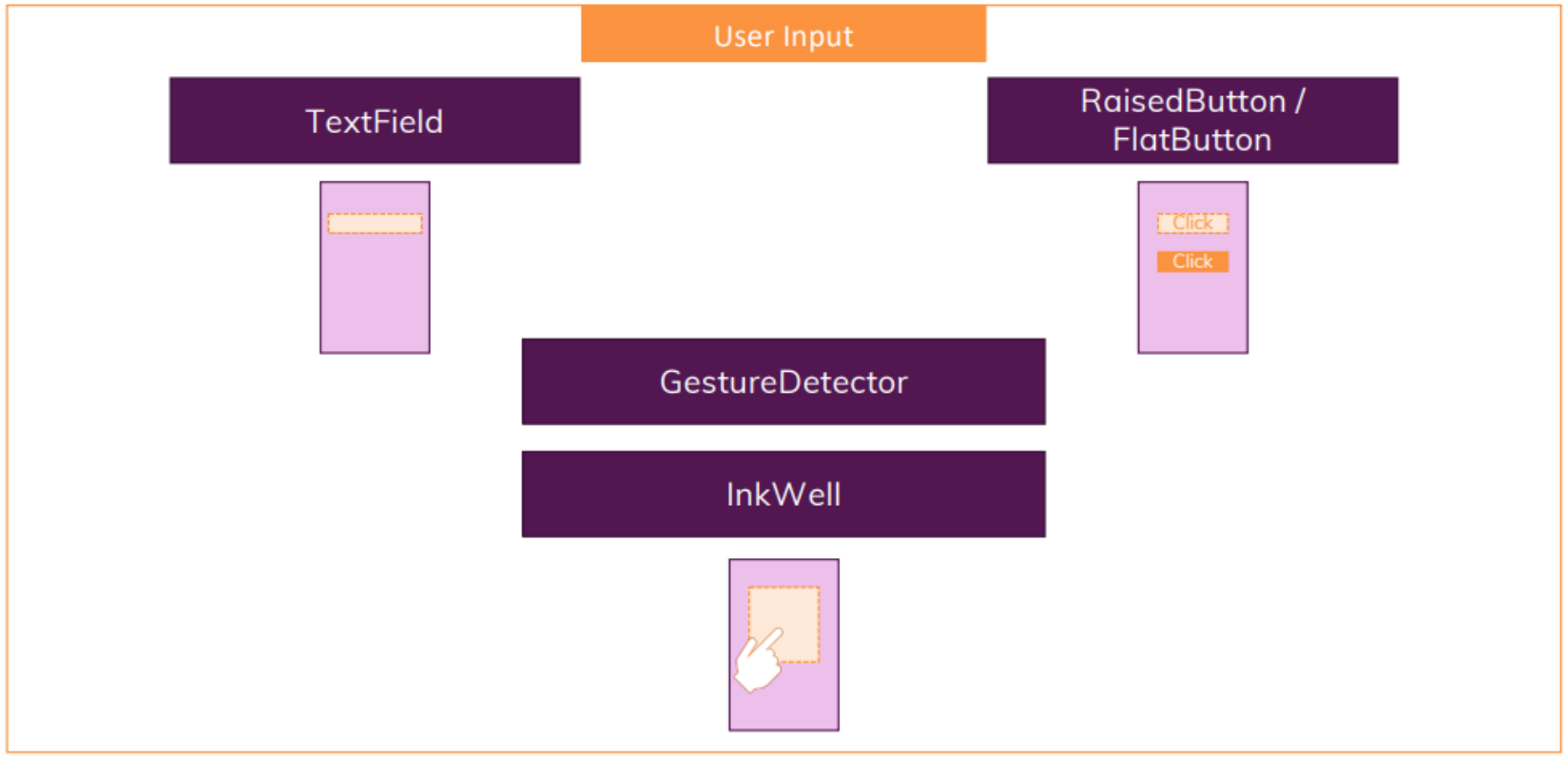
- Utilisé pour afficher une image à l'écran
- Prend en charge différentes sources (incluse dans l'application, image web, etc.)
- Peut être configuré pour s'adapter de différentes manières dans un conteneur enveloppant

Icon

- Affiche une icône à l'écran
- Flutter est livré avec de nombreuses icônes par défaut Material (Android) et iOS que vous pouvez utiliser
- Il existe également un widget IconButton() si vous avez besoin d'un bouton avec une icône

Introduction aux Widgets

Les Widgets les Plus Importants dans Flutter



Introduction aux Widgets

TextField

- Rendu d'un champ de texte éditable où l'utilisateur peut entrer (taper) des informations. De nombreuses options de configuration (ex : correction automatique, messages d'erreur, labels, styles).
- Supporte différents types de claviers (email, numéro, texte normal, ...).

RaisedButton / FlatButton / IconButton

- Différents styles de boutons qui gèrent les tapotements de l'utilisateur.
- Une fonction personnalisée qui doit être exécutée lors d'un tapotement doit être fournie.
- Peut être stylisé / personnalisé.

Introduction aux Widgets

GestureDetector / InkWell

- GestureDetector permet d'envelopper n'importe quel widget avec des écouteurs de gestes (ex : double tapotement, long tapotement).
- InkWell fait la même chose mais ajoute un effet de vague visuel lors des touches (cet effet peut être configuré).
- Vous pouvez construire vos propres boutons / widgets interactifs avec ces widgets.

Gestion des Listes

- Dans Flutter, les listes sont des composants essentiels pour afficher des collections d'éléments de manière efficace.
- Que ce soit pour afficher des textes, des images ou d'autres widgets, ListView et GridView sont les outils principaux pour la gestion des listes.

ListView

- ListView est un widget qui affiche une liste défilable d'éléments. Il peut être utilisé pour créer des listes statiques avec ListView ou des listes dynamiques avec ListView.builder.

Gestion des Listes

Listes Statique

- Ce code génère une liste simple avec des éléments statiques.

```
ListView(
  children: <Widget>[
    ListTile(
      leading: Icon(Icons.map),
      title: Text('Carte'),
    ),
    ListTile(
      leading: Icon(Icons.photo_album),
      title: Text('Album'),
    ),
    ListTile(
      leading: Icon(Icons.phone),
      title: Text('Téléphone'),
    ),
  ],
)
```

Gestion des Listes

- Ce code génère une liste d'éléments à partir d'une source de données dynamique (comme un tableau).

```
ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text('Item ${items[index]}'),
    );
  },
)
```

Avantages : ListView est particulièrement utile lorsque vous avez une grande quantité de données à afficher, car il ne rend que les éléments visibles à l'écran, ce qui améliore les performances.

Gestion des Listes

Scrollables et Items Dynamiques

- Vous pouvez créer des listes défilables de manière dynamique. Par exemple, si vous avez une grande liste de données provenant d'une API, vous pouvez les charger au fur et à mesure que l'utilisateur fait défiler la liste

Cet exemple montre une liste défilable d'éléments utilisateurs générée dynamiquement.

```
ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    return ListTile(
      leading: Icon(Icons.person),
      title: Text('Utilisateur ${items[index]}'),
    );
  },
)
```

Gestion des Listes

GridView

- GridView est utilisé pour afficher les éléments dans une grille. C'est idéal pour les affichages d'images ou autres contenus organisés en colonnes et lignes
- Ce code crée une grille de six conteneurs de différentes couleurs, disposés en trois colonnes.

```
GridView.count(
  crossAxisCount: 3,
  children: <Widget>[
    Container(color: Colors.red),
    Container(color: Colors.blue),
    Container(color: Colors.green),
    Container(color: Colors.yellow),
    Container(color: Colors.orange),
    Container(color: Colors.purple),
  ],
)
```

- Avantages : GridView est parfait pour les interfaces où les éléments doivent être uniformément répartis sur l'écran, comme les galeries de photos.

Navigation

- La navigation est un aspect crucial dans toute application mobile. Flutter utilise le Navigator pour gérer la navigation entre différents écrans (ou pages)

Navigator.push

- Navigator.push ajoute une nouvelle route à la pile de navigation, permettant de naviguer vers un nouvel écran

- Ce bouton permet de naviguer de la page actuelle vers une nouvelle page SecondPage.

```
ElevatedButton(
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => SecondPage()),
    );
  },
  child: Text('Aller à la deuxième page'),
)
```

Navigation

Navigator.pop :

- Navigator.pop enlève la route actuelle de la pile, revenant ainsi à l'écran précédent.

Ce bouton permet de revenir à la page précédente en retirant la route actuelle de la pile.

```
ElevatedButton(
  onPressed: () {
    Navigator.pop(context);
  },
  child: Text('Retour'),
)
```

Navigation

Passage de Données entre Écrans :

- Vous pouvez passer des données d'un écran à un autre en les incluant dans la méthode Navigator.push.
- Cet exemple montre comment envoyer des données à la page SecondPage lors de la navigation

```
ElevatedButton(
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => SecondPage(data: 'Hello from the first page'),
      ),
    );
  },
  child: Text('Envoyer des données'),
)
```

Navigation

Récupération des Données sur l'Écran Suivant

- Cet exemple montre comment envoyer des données à la page SecondPage lors de la navigation

```
class SecondPage extends StatelessWidget {
  final String data;

  SecondPage({required this.data});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Deuxième page')),
      body: Center(
        child: Text(data), // Affiche les données passées
      ),
    );
  }
}
```

Navigation

Retour d'Informations (passing data back) :

- Il est également possible de renvoyer des données à l'écran précédent lorsque vous revenez en arrière. Cela peut être utile pour des scénarios comme des formulaires de saisie ou la sélection d'éléments dans une liste
  - Dans cet exemple, lorsque vous revenez à la première page, un message contenant les données renvoyées par la deuxième page est affiché à l'aide d'un SnackBar.

```
// Première page
ElevatedButton(
  onPressed: () async {
    final result = await Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => SecondPage()),
    );

    // Utilisation du résultat retourné
    ScaffoldMessenger.of(context)
      .showSnackBar(SnackBar(content: Text('$result')));
  },
  child: Text('Aller à la deuxième page'),
);

// Deuxième page
ElevatedButton(
  onPressed: () {
    Navigator.pop(context, 'Données renvoyées de la deuxième page');
  },
  child: Text('Retourner des données'),
);
```

Navigation

Navigation avec des Routes Nommées

- En plus de Navigator.push et Navigator.pop, Flutter permet également d'utiliser des routes nommées pour naviguer entre les écrans de manière plus structurée.
- Configuration : Déclarer les routes nommées dans le widget MaterialApp.

```
MaterialApp(
  initialRoute: '/',
  routes: {
    '/': (context) => FirstPage(),
    '/second': (context) => SecondPage(),
  },
);
```

Navigation

Gestion de l'Historique de Navigation :

- Le Navigator de Flutter conserve un historique des routes (écrans) visitées, permettant de revenir en arrière ou d'accéder à une page spécifique à partir de cet historique.

  - Exemple : Supprimer toutes les routes jusqu'à une route spécifique.

```
ElevatedButton(
  onPressed: () {
    Navigator.of(context).pushNamedAndRemoveUntil(
      '/second', (Route<dynamic> route) => false);
  },
  child: Text('Aller à la deuxième page et effacer l\'historique'),
);
```

- Ce code permet de naviguer à une page spécifique et d'effacer l'historique de navigation, ce qui peut être utile après une opération de connexion ou d'inscription où vous ne voulez pas que l'utilisateur puisse revenir en arrière.

Création de Formulaires

Introduction aux Formulaires dans Flutter:

- Flutter offre plusieurs widgets et outils pour créer des formulaires robustes avec des validations.
- Utilisation des Champs de Texte, des Boutons, et des Validations
- **TextField et TextFormField** : TextField est un widget de base pour saisir du texte, tandis que TextFormField offre des fonctionnalités supplémentaires comme la validation dans un formulaire.

Création de Formulaires

```dart
class MyForm extends StatefulWidget {
  @override
  _MyFormState createState() => _MyFormState();
}

class _MyFormState extends State<MyForm> {
  final _formKey = GlobalKey<FormState>();
  String? _name;

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        children: <Widget>[
          TextFormField(
            decoration: InputDecoration(labelText: 'Name'),
            validator: (value) {
              if (value == null || value.isEmpty) {
                return 'Please enter your name';
              }
              return null;
            },
            onSaved: (value) {
              _name = value;
            },
          ),
          ElevatedButton(
            onPressed: () {
              if (_formKey.currentState?.validate() ?? false) {
                _formKey.currentState?.save();
                print('Name: $_name');
              }
            },
            child: Text('Submit'),
          ),
        ],
      ),
    );
  }
}
```

Exemple :

- Dans l'exemple ci-dessus, le widget Form contient un champ de texte TextFormField avec une validation. Lors de la soumission du formulaire, la méthode validate vérifie que tous les champs respectent les règles de validation, et la méthode save enregistre les données.

Gestion de l'État

- La gestion de l'état est un concept clé dans le développement d'applications Flutter => mise à jour et partage des données dans une application

Explication des Différents Types de Gestion d'État (Local, Global)

- Gestion d'État Locale : L'état est maintenu au niveau d'un seul widget. C'est le type de gestion d'état le plus simple et souvent utilisé avec setState.
- **setState()** : C'est la méthode la plus simple pour gérer l'état local d'un widget. Elle est souvent utilisée pour des cas simples où l'état ne doit être partagé que dans un seul widget.
- Gestion d'État Globale : L'état est partagé entre plusieurs widgets dans l'application. Cela peut être réalisé à l'aide de InheritedWidget, Provider, ou d'autres solutions comme Riverpod.

Gestion de l'État

- Exemple Etat local avec Setstate()

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Counter: $_counter'),
        ElevatedButton(
          onPressed: _incrementCounter,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

Gestion de l'État

- InheritedWidget : Cette classe permet de partager l'état entre un widget parent et ses descendants sans avoir à passer l'état explicitement à chaque widget.

```
class MyInheritedWidget extends InheritedWidget {
  final int data;

  MyInheritedWidget({required this.data, required Widget child}) : super(child: child

  @override
  bool updateShouldNotify(MyInheritedWidget oldWidget) => data != oldWidget.data;

  static MyInheritedWidget? of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<MyInheritedWidget>();
  }
}

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
```

Gestion de l'État

```
  @override
  Widget build(BuildContext context) {
    return MyInheritedWidget(
      data: _counter,
      child: Column(
        children: <Widget>[
          ElevatedButton(
            onPressed: _incrementCounter,
            child: Text('Increment'),
          ),
          const ChildWidget(),
        ],
      ),
    );
  }
}

class ChildWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final inheritedWidget = MyInheritedWidget.of(context);
    return Text('Counter: ${inheritedWidget?.data ?? 0}');
  }
}
```

Gestion de l'État

- Provider : Provider est une solution populaire et flexible pour la gestion d'état globale. Il simplifie le partage d'état entre les widgets et gère également l'écoute des changements d'état.

- Exemple

```
class CounterProvider with ChangeNotifier {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    _counter++;
    notifyListeners();
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterProvider(),
      child: MaterialApp(
        home: Scaffold(
          appBar: AppBar(title: Text('Provider Example')),
          body: CounterScreen(),
        ),
      ),
    );
  }
}
```

Gestion de l'État

```
class CounterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterProvider = Provider.of<CounterProvider>(context);
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text('Counter: ${counterProvider.counter}'),
        ElevatedButton(
          onPressed: counterProvider.increment,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

Gestion Avancée de l'État avec Provider

Introduction à Provider

- Scalabilité : Pour une application complexe, l'état est géré de manière centralisée et efficace. Provider est une solution de gestion d'état qui facilite cette tâche en permettant de partager et de manipuler l'état à travers toute l'application.
- Réactivité : Avec Provider, les widgets réagissent automatiquement aux changements d'état, garantissant une mise à jour de l'interface utilisateur en temps réel.
- Séparation des préoccupations : En utilisant Provider, vous pouvez séparer la logique d'affaires de la présentation de l'interface utilisateur, rendant votre code plus propre et plus facile à maintenir.

Gestion Avancée de l'État avec Provider

Configuration de Provider dans un projet Flutter

- Pour utiliser Provider, il faut d'abord l'ajouter à votre projet. Ajoutez provider à votre fichier
- pubspec.yaml :

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.0
```

Gestion Avancée de l'État avec Provider

- Configurez Provider au niveau supérieur de votre application pour qu'il puisse être accessible à tous les widgets enfants :

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => TaskProvider()),
      ],
      child: MyApp(),
    ),
  );
}
```

Gestion Avancée de l'État avec Provider

- Exemple de partage d'état : Supposons que vous avez une classe TaskProvider qui gère une liste de tâches dans une application ToDo :

- Explication : TaskProvider contient une liste de tâches et des méthodes pour ajouter ou supprimer des tâches. notifyListeners() est appelé pour informer tous les widgets qui écoutent ce Provider que l'état a changé, afin qu'ils puissent se reconstruire avec les nouvelles données.

```
class TaskProvider with ChangeNotifier {
  List<String> _tasks = [];

  List<String> get tasks => _tasks;

  void addTask(String task) {
    _tasks.add(task);
    notifyListeners();
  }

  void removeTask(String task) {
    _tasks.remove(task);
    notifyListeners();
  }
}
```

Gestion Avancée de l'État avec Provider

Mise en pratique avec un projet d'application ToDo

- Liste des tâches : Utilisation de ListView.builder pour afficher les tâches.
- Ajout de tâches : Un TextField et un FloatingActionButton pour ajouter des nouvelles tâches.
- Suppression de tâches : Chaque tâche peut être supprimée en utilisant un Dismissible widget.

Gestion Avancée de l'État avec Provider

- Exemple de mise en pratique

```
class TodoApp extends StatelessWidget
  @override
  Widget build(BuildContext context)
    return ChangeNotifierProvider(
      create: (_) => TaskProvider(),
      child: MaterialApp(
        home: TaskScreen(),
      ),
    );
  }
}
```

```
class TaskScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final taskProvider = Provider.of<TaskProvider>(context);

    return Scaffold(
      appBar: AppBar(
        title: Text('ToDo App'),
      ),
      body: ListView.builder(
        itemCount: taskProvider.tasks.length,
        itemBuilder: (context, index) {
          final task = taskProvider.tasks[index];
          return Dismissible(
            key: Key(task),
            onDismissed: (direction) {
              taskProvider.removeTask(task);
            },
            child: ListTile(
              title: Text(task),
            ),
          );
        },
      ),
```

Gestion Avancée de l'État avec Provider

- Exemple de mise en pratique

```
        floatingActionButton: FloatingActionButton(
          onPressed: () {
            // Logic to add a new task
            taskProvider.addTask('Nouvelle Tâche');
          },
          child: Icon(Icons.add),
        ),
      );
    }
}
```

- Explication : Dans cet exemple, TaskScreen écoute les modifications de TaskProvider. Lorsqu'une nouvelle tâche est ajoutée ou supprimée, l'interface utilisateur se met automatiquement à jour.

Les Animations et les Transitions

Introduction aux Animations

- Les animations rendent une application plus dynamique et agréable à utiliser. Elles permettent d'attirer l'attention de l'utilisateur, d'améliorer l'expérience utilisateur, et de rendre les interactions plus intuitives.
- Flutter propose un riche ensemble de widgets et de classes pour créer des animations. Cela inclut des widgets prédéfinis pour des animations simples ainsi que des API pour des animations plus complexes.
- Création d'animations simples avec AnimatedContainer, AnimatedOpacity, etc. :
- AnimatedContainer : Ce widget est similaire au widget Container mais permet d'animer ses propriétés, comme la couleur, la taille, la marge, etc., lorsque l'état change.

Les Animations et les Transitions

```dart
class AnimatedContainerExample extends StatefulWidget {
  @override
  _AnimatedContainerExampleState createState() => _AnimatedContainerExampleState();
}

class _AnimatedContainerExampleState extends State<AnimatedContainerExample> {
  bool _isLarge = false;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('AnimatedContainer Example')),
      body: Center(
        child: AnimatedContainer(
          width: _isLarge ? 200.0 : 100.0,
          height: _isLarge ? 200.0 : 100.0,
          color: _isLarge ? Colors.blue : Colors.red,
          duration: Duration(seconds: 1),
          curve: Curves.easeInOut,
          child: GestureDetector(
            onTap: () {
              setState(() {
                _isLarge = !_isLarge;
              });
            },
          ), // GestureDetector
        ), // AnimatedContainer
      ), // Center
    ); // Scaffold
  }
}
```

- Explication : Ici, AnimatedContainer change de taille et de couleur lorsque l'utilisateur clique dessus. L'animation est définie par une Duration et une Curve.

Les Animations et les Transitions

- AnimatedOpacity : Ce widget permet d'animer l'opacité d'un widget enfant, ce qui peut être utilisé pour créer des effets de fondu.

- Explication : AnimatedOpacity permet de passer de l'opacité complète (1.0) à l'opacité nulle (0.0) en cliquant sur le carré vert.

```dart
class AnimatedOpacityExample extends StatefulWidget {
  @override
  _AnimatedOpacityExampleState createState() => _AnimatedOpacityExampleState();
}

class _AnimatedOpacityExampleState extends State<AnimatedOpacityExample> {
  double _opacity = 1.0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('AnimatedOpacity Example')),
      body: Center(
        child: GestureDetector(
          onTap: () {
            setState(() {
              _opacity = _opacity == 1.0 ? 0.0 : 1.0;
            });
          },
          child: AnimatedOpacity(
            opacity: _opacity,
            duration: Duration(seconds: 2),
            child: Container(
              width: 200,
              height: 200,
              color: Colors.green,
            ), // Container
          ), // AnimatedOpacity
        ), // GestureDetector
      ), // Center
    ); // Scaffold
  }
}
```

Les Animations et les Transitions

Transitions entre Écrans

- Flutter permet d'animer les transitions entre les différentes pages ou écrans d'une application. En personnalisant la navigation, on peut rendre les transitions plus fluides et visuellement attractives.

Personnalisation des transitions

- Utilisation de PageRouteBuilder : Pour créer des transitions personnalisées entre les écrans, on peut utiliser la classe PageRouteBuilder.

Les Animations et les Transitions

- Explication : Ici, FadeTransition est utilisé pour créer un effet de fondu lors de la navigation vers SecondPage. Le transitionsBuilder permet de personnaliser l'animation de transition.

```
Navigator.push(
  context,
  PageRouteBuilder(
    pageBuilder: (context, animation, secondaryAnimation) => SecondPage(),
    transitionsBuilder: (context, animation, secondaryAnimation, child) {
      return FadeTransition(
        opacity: animation,
        child: child,
      );
    },
  ),
);
```