

TEXT WIDGET

INTRODUCTION AU TEXT WIDGET

Le `Text` widget dans Flutter est utilisé pour afficher une simple chaîne de caractères sur l'écran. Il est l'un des widgets les plus couramment utilisés pour montrer du texte dans les applications Flutter.

SYNTAXE DE BASE DU TEXT WIDGET

```
Text('Votre texte ici',  
    style: TextStyle(  
        fontSize: 20.0  
    ),  
)
```

PROPRIÉTÉS DU TEXT WIDGET

- **data** : le texte à afficher.
- **style** : le style du texte.
- **textAlign** : l'alignement du texte.
- **maxLines** : le nombre maximum de lignes pour le texte.
- **overflow** : la gestion du débordement de texte.

STYLE DU TEXTE DANS TEXT WIDGET

```
Text('Bonjour Flutter!',  
    style: TextStyle(  
        fontSize: 24.0,  
        fontWeight: FontWeight.bold,  
        color: Colors.blue,  
    ),  
)
```

ALIGNEMENT DU TEXTE DANS TEXT WIDGET

```
Text('Aligné au centre',  
     textAlign: TextAlign.center,  
)
```

ROW WIDGET

STRUCTURE DE BASE D'UN ROW WIDGET

Un Row widget organise ses enfants en ligne horizontale. Voici la structure de base :

```
Row(  
  children: <Widget>[  
    // Widgets enfants  
  ],  
)
```

ALIGNEMENT DES ÉLÉMENTS DANS UN ROW

Les éléments dans un **Row** peuvent être alignés verticalement et horizontalement. Par défaut, ils sont alignés au centre verticalement.

UTILISATION DE MAINAXISALIGNMENT ET CROSSAXISALIGNMENT

```
Row(  
    mainAxisAlignment: MainAxisAlignment.start, // Alignement horizontal  
    crossAxisAlignment: CrossAxisAlignment.center, // Alignement vertical  
    children: <Widget>[  
        // Widgets enfants  
    ],  
)
```

AJOUT D'ESPACEMENT ENTRE LES ÉLÉMENTS AVEC SIZEDBOX ET SPACER

Pour ajouter de l'espace entre les éléments dans un Row :

```
Row(  
  children: <Widget>[  
    Widget1(),  
    SizedBox(width: 20), // Espace fixe  
    Spacer(),         // Espace flexible  
    Widget2(),  
  ],  
)
```

COLUMN WIDGET

STRUCTURE DU COLUMN WIDGET

Le `Column` widget permet de disposer des widgets enfants de manière verticale. Voici sa structure de base :

```
Column(  
  children: <Widget>[  
    Widget1,  
    Widget2,  
    Widget3,  
    // Ajoutez d'autres widgets ici  
  ],  
)
```

PROPRIÉTÉS PRINCIPALES DU COLUMN WIDGET

- **mainAxisAlignment**: Alignement vertical des enfants.
- **crossAxisAlignment**: Alignement horizontal des enfants.
- **mainAxisSize**: Taille du contenu selon l'axe principal.
- **verticalDirection**: Direction des enfants (de haut en bas ou de bas en haut).

ALIGNEMENT VERTICAL DANS LE COLUMN WIDGET

Propriété mainAxisAlignment	Description
start	Alignement en haut
end	Alignement en bas
center	Alignement au centre
spaceBetween	Espace égal entre les enfants
spaceAround	Espace autour de chaque enfant
spaceEvenly	Espace égal entre les bords et enfants

GESTION DE L'ESPACE ENTRE LES ÉLÉMENTS

Pour gérer l'espace entre les éléments dans un `Column`, utilisez les propriétés de `mainAxisAlignment` :

- `spaceBetween` : Espace égal entre chaque enfant.
- `spaceAround` : Espace autour de chaque enfant.
- `spaceEvenly` : Espace égal entre les enfants et aussi autour de tous les enfants.

CONTAINER WIDGET

PROPRIÉTÉS DE BASE DU CONTAINER

- **child** : Widget à afficher à l'intérieur du Container.
- **width** : Largeur du Container.
- **height** : Hauteur du Container.
- **padding** : Espace intérieur entre les bords du Container et son enfant.
- **margin** : Espace extérieur autour du Container.
- **alignment** : Alignement du widget enfant à l'intérieur du Container.

BORDURES ET MARGES

- **border** : Définit la bordure du Container. Utilise `Border.all()` pour une bordure uniforme.
- Exemple de bordure :

```
border: Border.all(color: Colors.blue, width: 2)
```

- **margin** : Espace autour du Container. Utilise `EdgeInsets` pour spécifier.
- Exemple de marge :

```
margin: EdgeInsets.all(10) // Marge sur tous les côtés
```

ALIGNEMENT ET POSITIONNEMENT

- alignment : Contrôle où le child est positionné dans le Container.
- Valeurs possibles : Alignment.center, Alignment.topLeft, Alignment.bottomRight, etc.
- Exemple :

```
alignment: Alignment.center
```

- Utilisation de Transform pour des positionnements avancés.

COULEUR ET DÉCORATION

- **color** : Définit la couleur de fond du Container.
- **decoration** : Utilisé pour la décoration avancée comme les gradients, les images de fond, etc.
- Exemple de décoration avec gradient :

```
decoration: BoxDecoration(  
    gradient: LinearGradient(colors: [Colors.blue, Colors.green])  
)
```

UTILISATION DE BOXCONSTRAINTS

- `constraints` : Permet de spécifier les contraintes pour les dimensions du Container.
- Types de contraintes : `BoxConstraints.expand()`, `BoxConstraints.tightFor(width: 100, height: 100)`, etc.
- Exemple d'utilisation :

```
constraints: BoxConstraints.tightFor(width: 100, height: 100)
```

IMAGE WIDGET

INTRODUCTION À IMAGE WIDGET

Flutter fournit le widget **Image** pour afficher des images. Il peut charger des images à partir de différentes sources telles que des fichiers locaux et des URL distantes. Utiliser **Image** améliore l'interface utilisateur en intégrant des visuels attractifs.

UTILISATION DE ASSETIMAGE

```
Image(  
    image: AssetImage( 'chemin/vers/image.png' )  
)
```

AssetImage est utilisé pour charger des images stockées dans les ressources de l'application. Indiquez le chemin relatif à partir du dossier des ressources.

UTILISATION DE NETWORKIMAGE

```
Image(  
  image: NetworkImage('https://exemple.com/image.png')  
)
```

NetworkImage charge des images depuis Internet. Fournissez l'URL de l'image que vous souhaitez afficher.

PARAMÈTRES CLÉS DE IMAGE WIDGET (FIT, ALIGNMENT)

- **fit:** Définit comment une image doit être insérée dans l'espace alloué.
 - BoxFit.fill: Remplit tout l'espace, peut déformer l'image.
 - BoxFit.contain: Garde les proportions, peut laisser des espaces vides.
- **alignment:** Positionne l'image dans son conteneur.
 - Alignment.center: Centre l'image.
 - Alignment.topLeft: Alignement en haut à gauche.

GESTION DES ERREURS D'IMAGE

Pour gérer les erreurs lors du chargement des images, utilisez `errorBuilder` :

```
Image.network(  
    'https://exemple.com/image.png',  
    errorBuilder: (BuildContext context, Object exception, StackTrace? stackTrace) {  
        return Text('Impossible de charger l\'image');  
    }  
)
```

Cela affiche un texte d'erreur si l'image ne peut pas être chargée.

BUTTON WIDGET

TYPES DE BOUTONS DANS FLUTTER

Flutter propose plusieurs types de boutons :

- **ElevatedButton** : bouton avec un effet de relief.
- **FlatButton** (obsolète, remplacé par **TextButton**) : bouton sans relief.
- **IconButton** : bouton iconique sans texte.
- **FloatingActionButton** : bouton circulaire émergeant.
- **OutlinedButton** : bouton avec un contour.

CRÉATION D'UN BOUTON SIMPLE

Pour créer un bouton simple avec Flutter, utilisez le widget `ElevatedButton` :

```
ElevatedButton(  
  onPressed: () {  
    // Action à exécuter  
  },  
  child: Text('Cliquez ici'),  
)
```

PERSONNALISATION DES BOUTONS

Personnaliser un bouton dans Flutter peut inclure :

- Couleur : style: ElevatedButton.styleFrom(primary: Colors.blue)
- Padding : style: ElevatedButton.styleFrom(padding: EdgeInsets.all(16))
- Bordure : style: ElevatedButton.styleFrom(shape:
RoundedRectangleBorder(borderRadius: BorderRadius.circular(30)))

GESTION DES ÉVÉNEMENTS SUR LES BOUTONS

Pour gérer les événements sur un bouton Flutter, définissez la fonction `onPressed` :

```
ElevatedButton(  
  onPressed: () {  
    // Code à exécuter quand le bouton est pressé  
  },  
  child: Text('Appuyez-moi'),  
)
```

SCAFFOLD WIDGET

STRUCTURE DE BASE D'UN SCAFFOLD

Le **Scaffold** est un widget dans Flutter qui fournit une structure de base pour l'interface utilisateur. Il offre un cadre pour ajouter des barres d'applications, des barres de navigation, des tiroirs et plus encore.

```
Scaffold(  
  appBar: AppBar(),  
  body: Widget(),  
  floatingActionButton: FloatingActionButton(),  
  drawer: Drawer(),  
  bottomNavigationBar: BottomNavigationBar(),  
)
```

APPBAR

L'AppBar est un widget situé en haut de l'écran dans un Scaffold. Il est généralement utilisé pour afficher le titre de l'application, des actions ou des boutons de navigation.

```
AppBar(  
    title: Text('Titre de l'AppBar'),  
    actions: <Widget>[  
        IconButton(icon: Icon(Icons.search), onPressed: () {}),  
    ],  
)
```

FLOATINGACTIONBUTTON

Le **FloatingActionButton** est un bouton circulaire qui flotte généralement au-dessus du contenu pour promouvoir une action principale dans l'application.

```
FloatingActionButton(  
    onPressed: () {},  
    child: Icon(Icons.add),  
)
```

DRAWER

Le **Drawer** est un panneau coulissant qui s'ouvre latéralement pour afficher des options de navigation ou d'autres contenus dans une application Flutter.

```
Drawer(  
  child: ListView(  
    children: <Widget>[  
      DrawerHeader(child: Text('En-tête du Drawer')),  
      ListTile(title: Text('Item 1')),  
    ],  
  ),  
)
```

BOTTOMNAVIGATIONBAR

Le `BottomNavigationBar` est utilisé pour la navigation entre les vues de haut niveau d'une application via une barre située en bas de l'écran.

```
BottomNavigationBar(  
  items: const <BottomNavigationBarItem>[  
    BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Accueil'),  
    BottomNavigationBarItem(icon: Icon(Icons.business), label: 'Business'),  
  ],  
)
```

CORPS DU SCAFFOLD (BODY)

Le **body** est la partie principale du **Scaffold** où le contenu principal de l'écran est affiché.

```
Scaffold(  
    body: Center(  
        child: Text('Contenu principal ici'),  
    ),  
)
```

LISTVIEW WIDGET

INTRODUCTION AU LISTVIEW

Le **ListView** est un widget scrollable dans Flutter qui permet d'afficher une liste d'éléments de manière verticale ou horizontale. Il est utilisé pour afficher une liste dynamique d'éléments tels que des textes, des images, ou des widgets personnalisés.

CRÉATION D'UN LISTVIEW SIMPLE

```
ListView(  
  children: <Widget>[  
    ListTile(title: Text('Element 1')),  
    ListTile(title: Text('Element 2')),  
    ListTile(title: Text('Element 3')),  
  ],  
)
```

LISTVIEW.BUILDER

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

GESTION DE LA SCROLLABILITÉ

- **scrollDirection**: Définit la direction du défilement (`Axis.vertical` ou `Axis.horizontal`).
- **reverse**: Inverse l'ordre de défilement des éléments.
- **controller**: Contrôle le défilement avec un `ScrollController`.

PERSONNALISATION DES ÉLÉMENTS DU LISTVIEW

- Utilisation de différents types de widgets comme `ListTile`, `Container`, ou widgets personnalisés.
- Modification de l'apparence avec `padding`, `margin`, et `decoration`.
- Interaction avec les éléments via `onTap`, `onLongPress`, etc.

STACK WIDGET

INTRODUCTION AU STACK WIDGET

Le Stack Widget dans Flutter permet de superposer des widgets les uns sur les autres. Il est souvent utilisé pour créer des interfaces utilisateur complexes et personnalisées, en plaçant des éléments à des positions arbitraires.

STRUCTURE DE BASE D'UN STACK

```
Stack(  
  children: <Widget>[  
    // Liste des widgets à superposer  
  ],  
)
```

- **children** est une liste de widgets qui seront empilés les uns sur les autres, le premier widget de la liste étant au bas de la pile.

ALIGNEMENT DES ÉLÉMENTS DANS UN STACK

```
Stack(  
  alignment: Alignment.center, // Alignement central par défaut  
  children: <Widget>[  
    // Widgets  
  ],  
)
```

- **alignment** détermine comment les enfants non positionnés sont alignés dans le Stack.

UTILISATION DE POSITIONED POUR PLACER LES ÉLÉMENTS

```
Stack(  
  children: <Widget>[  
    Positioned(  
      top: 10,  
      left: 10,  
      child: Text('En haut à gauche'),  
    ),  
    // Autres widgets positionnés  
  ],  
)
```

- **Positioned** permet de placer précisément un widget à l'intérieur du Stack en spécifiant les coordonnées.

CAS PRATIQUES D'UTILISATION DU STACK WIDGET

1. Superposition d'une image et d'un texte pour créer un effet de légende.
 2. Création de badges ou d'indicateurs sur des icônes.
 3. Mise en place d'éléments flottants sur une carte ou une image.
- Ces exemples montrent comment le Stack peut être utilisé pour enrichir l'interface utilisateur.

FORM WIDGET

STRUCTURE D'UN FORM WIDGET

Un **Form** widget dans Flutter est utilisé pour regrouper plusieurs champs de formulaire ensemble. Voici sa structure de base :

```
Form(  
  key: _formKey,  
  child: Column(  
    children: <Widget>[  
      // Ajouter des TextFormField ici  
    ],  
  ),  
)
```

UTILISATION DE TEXTFORMFIELD

Le `TextField` est utilisé pour entrer du texte dans un formulaire. Exemple d'utilisation :

```
TextField(  
    decoration: InputDecoration(labelText: 'Entrez votre nom'),  
    validator: (value) {  
        if (value.isEmpty) {  
            return 'Veuillez entrer du texte';  
        }  
        return null;  
    },  
)
```

GESTION DES CLÉS DE FORMULAIRE

Les clés de formulaire (`FormKey`) permettent de contrôler l'état du formulaire, comme la validation des données ou la réinitialisation du formulaire. Exemple :

```
final _formKey = GlobalKey<FormState>();
```

VALIDATION DES DONNÉES SAISIES

La validation est réalisée à l'aide de la propriété `validator` de `TextField`. Elle permet de vérifier que les données entrées sont correctes :

```
validator: (value) {  
  if (value.isEmpty) {  
    return 'Ce champ ne peut pas être vide';  
  }  
  return null;  
}
```

SOUMISSION DU FORMULAIRE

Pour soumettre un formulaire, utilisez la méthode `validate` de la clé du formulaire. Si la validation réussit, traitez les données :

```
if (_formKey.currentState.validate()) {  
    // Traiter les données du formulaire  
}
```

SLIDER WIDGET

FONCTIONNALITÉ DU SLIDER WIDGET

Le Slider Widget permet à l'utilisateur de sélectionner une valeur dans une plage en faisant glisser un curseur horizontal. Utilisé pour ajuster des valeurs comme le volume, la luminosité, ou tout paramètre nécessitant une sélection continue entre deux extrêmes.

CRÉATION D'UN SLIDER WIDGET

```
Slider(  
  value: _currentValue,  
  min: 0,  
  max: 100,  
  onChanged: (double value) {  
    setState(() {  
      _currentValue = value;  
    });  
  },  
)
```

PERSONNALISATION DU SLIDER WIDGET

```
Slider(  
  value: _currentValue,  
  min: 0,  
  max: 100,  
  divisions: 5,  
  label: '${_currentValue.round()}',  
  activeColor: Colors.blue,  
  inactiveColor: Colors.grey,  
  onChanged: (double value) {  
    setState(() {  
      _currentValue = value;  
    });  
  },  
)
```

GESTION DES ÉVÉNEMENTS DU SLIDER WIDGET

```
Slider(  
  value: _currentValue,  
  min: 0,  
  max: 100,  
  onChanged: (double value) {  
    setState(() {  
      _currentValue = value;  
    });  
    print("Slider Value: $value");  
  },  
)
```

SWITCH WIDGET

FONCTIONNEMENT DU SWITCH WIDGET

Le Switch Widget dans Flutter est utilisé pour basculer entre deux états : activé ou désactivé. Il est communément utilisé dans les paramètres pour activer/désactiver des fonctionnalités.

CRÉATION D'UN SWITCH WIDGET

```
Switch(  
    value: isSwitched,  
    onChanged: (value) {  
        setState(() {  
            isSwitched = value;  
        });  
    },  
)
```

- **value** : état actuel du Switch (true ou false).
- **onChanged** : fonction appelée à chaque changement d'état.

GESTION DE L'ÉTAT DU SWITCH

```
bool isSwitched = false; // État initial du Switch

Switch(
  value: isSwitched,
  onChanged: (bool newValue) {
    setState(() {
      isSwitched = newValue;
    });
  },
)
```

- Utilisation de `setState` pour mettre à jour l'état du widget.

PERSONNALISATION DU SWITCH WIDGET

```
Switch(  
    value: isSwitched,  
    onChanged: (value) {...},  
    activeColor: Colors.green,  
    inactiveThumbColor: Colors.red,  
    activeTrackColor: Colors.lightGreen,  
    inactiveTrackColor: Colors.grey,  
)
```

- **activeColor** : couleur du bouton lorsque le Switch est activé.
- **inactiveThumbColor** : couleur du bouton lorsque désactivé.
- **activeTrackColor** : couleur de la piste lorsque activé.
- **inactiveTrackColor** : couleur de la piste lorsque désactivé.

APPBAR WIDGET

STRUCTURE DE L'APPBAR

L'AppBar de Flutter est une barre d'application typiquement utilisée pour des actions et des titres de navigation. Voici sa structure de base :

```
AppBar(  
  title: Widget,  
  leading: Widget,  
  actions: <Widget>[],  
  bottom: PreferredSizeWidget,  
)
```

PROPRIÉTÉS DE L'APPBAR

- **title:** Le titre affiché dans l'AppBar.
- **leading:** Un widget placé avant le titre.
- **actions:** Une liste de widgets affichés à la droite du titre.
- **backgroundColor:** Couleur de fond de l'AppBar.
- **elevation:** L'ombre portée sous l'AppBar.

EXEMPLE SIMPLE D'APPBAR

```
AppBar(  
    title: Text('Mon Application'),  
    actions: <Widget>[  
        IconButton(  
            icon: Icon(Icons.search),  
            onPressed: () {},  
        ),  
    ],  
)
```

INTERACTION AVEC L'APPBAR

Les interactions courantes avec l'AppBar incluent :

- Utilisation de `IconButton` dans `actions` pour ajouter des fonctionnalités.
- Modification de `leading` pour personnaliser le widget de navigation.
- Utilisation de `onPressed` dans `IconButton` pour définir des actions spécifiques.

FLOATINGACTIONBUTTON WIDGET

DÉFINITION DU FLOATINGACTIONBUTTON

Le FloatingActionButton est un bouton circulaire flottant dans Flutter. Il est généralement placé en bas à droite de l'écran. Ce widget est utilisé pour promouvoir une action principale dans l'application.

UTILISATION ET CAS D'USAGE

- Action principale : Utilisé pour l'action la plus fréquente de l'application (ex : ajouter, créer, partager).
- Positionnement : Se trouve généralement en bas à droite pour une accessibilité facile.
- Visibilité : Flotte au-dessus du contenu pour une visibilité constante.

PROPRIÉTÉS CLÉS DU FLOATINGACTIONBUTTON

- `onPressed`: Définit la fonction à exécuter lors de l'appui sur le bouton.
- `child`: Widget à afficher à l'intérieur du bouton, souvent une icône.
- `backgroundColor`: Couleur de fond du bouton.
- `elevation`: L'ombre portée autour du bouton.

PERSONNALISATION DU FLOATINGACTIONBUTTON

- Icônes : Choix de l'icône à afficher (ex : Icons . add).
- Couleurs : Modification de la couleur de fond et de l'icône.
- Taille : Ajustement de la taille via `mini` pour une version plus petite.
- Animation : Animation lors de l'apparition ou de la disparition.

INTERACTION AVEC FLOATINGACTIONBUTTON (ACTIONS SUR APPUI)

```
FloatingActionButton(  
    onPressed: () {  
        // Action à réaliser  
    },  
    child: Icon(Icons.add),  
    backgroundColor: Colors.green,  
)
```

DRAWER WIDGET

CRÉATION D'UN DRAWER WIDGET

Pour créer un Drawer dans Flutter, utilisez le widget `Drawer`:

```
Drawer(  
  child: ListView(  
    // Ajout des enfants ici  
  ),  
)
```

STRUCTURE ET COMPOSITION D'UN DRAWER

Un Drawer typique contient :

- Un en-tête (`DrawerHeader` ou autre widget)
- Une liste d'éléments (`ListView` avec des `ListTile`)

```
Drawer(  
  child: ListView(  
    children: [  
      DrawerHeader(child: Text('En-tête')),  
      ListTile(title: Text('Item 1')),  
      ListTile(title: Text('Item 2')),  
    ],  
  ),  
)
```

INTÉGRATION DU DRAWER DANS UNE APPLICATION FLUTTER

Intégrez le Drawer dans un `Scaffold` pour l'utiliser dans une application :

```
Scaffold(  
    appBar: AppBar(title: Text('Page Title')),  
    drawer: Drawer(  
        // Contenu du Drawer ici  
    ),  
)
```

PERSONNALISATION DU DRAWER (COULEUR, TAILLE, CONTENU)

Personnalisez le Drawer en modifiant :

- Couleur de fond : `backgroundColor`
- Taille : Utilisez `width` pour contrôler la largeur
- Contenu : Ajoutez ou modifiez des widgets dans `ListView`

```
Drawer(  
  backgroundColor: Colors.blue,  
  child: ListView(  
    children: [  
      DrawerHeader(child: Text('En-tête')),  
      ListTile(title: Text('Personnalisé')),  
    ],  
  ),  
)
```

INTERACTION AVEC LES ÉLÉMENTS DU DRAWER (NAVIGATION, CLICS)

Gérez les interactions avec les éléments du Drawer :

- Utilisez `onTap` dans `ListTile` pour gérer les clics
- Naviguez vers d'autres pages ou effectuez des actions

```
 ListTile(  
  title: Text('Navigation'),  
  onTap: () {  
    Navigator.push(context, MaterialPageRoute(builder: (context) => NewPage()));  
  },  
)
```

BOTTOMNAVIGATIONBAR WIDGET

INTRODUCTION AU BOTTOMNAVIGATIONBAR

BottomNavigationBar est un widget dans Flutter utilisé pour naviguer entre les vues au niveau de la partie inférieure de l'écran. Il permet d'afficher entre trois et cinq éléments de navigation, chacun représenté par une icône et un texte facultatif.

STRUCTURE DE BOTTOMNAVIGATIONBAR

```
BottomNavigationBar(  
    items: <BottomNavigationBarItem>[  
        BottomNavigationBarItem(  
            icon: Icon(Icons.home),  
            label: 'Accueil',  
        ),  
        BottomNavigationBarItem(  
            icon: Icon(Icons.business),  
            label: 'Business',  
        ),  
    ],  
)
```

UTILISATION DE BOTTOMNAVIGATIONBARITEM

Chaque item du `BottomNavigationBar` est un `BottomNavigationBarItem` qui comprend principalement un `icon` et un `label`. Exemple d'utilisation :

```
BottomNavigationBarItem(  
  icon: Icon(Icons.home),  
  label: 'Accueil'  
)
```

PERSONNALISATION DU BOTTOMNAVIGATIONBAR

Pour personnaliser le `BottomNavigationBar`, vous pouvez modifier les propriétés comme `backgroundColor`, `selectedItemColor`, `unselectedItemColor`, etc.

```
BottomNavigationBar(  
    backgroundColor: Colors.blue,  
    selectedItemColor: Colors.white,  
    unselectedItemColor: Colors.grey,  
)
```

GESTION DES ÉTATS AVEC BOTTOMNAVIGATIONBAR

La gestion des états se fait en utilisant la propriété `currentIndex` pour contrôler l'item actif et `onTap` pour gérer les actions de l'utilisateur :

```
int _selectedIndex = 0;

BottomNavigationBar(
  currentIndex: _selectedIndex,
  onTap: (int index) {
    setState(() {
      _selectedIndex = index;
    });
  },
)
```

TABBAR WIDGET

FONCTIONNEMENT DU TABBAR WIDGET

Le TabBar Widget dans Flutter permet de créer des onglets horizontaux pour la navigation. Il est généralement utilisé en combinaison avec un `TabBarView` pour afficher différents widgets en fonction de l'onglet sélectionné. Chaque onglet peut contenir un texte, une icône, ou les deux.

CRÉATION D'UN TABBAR

```
TabBar(  
  tabs: [  
    Tab(icon: Icon(Icons.directions_car)),  
    Tab(icon: Icon(Icons.directions_transit)),  
    Tab(icon: Icon(Icons.directions_bike)),  
  ],  
)
```

- **tabs:** Liste des widgets Tab à afficher.

CONFIGURATION DES ONGLETS

```
TabController _tabController;  
  
void initState() {  
    super.initState();  
    _tabController = TabController(vsync: this, length: 3);  
}
```

- Initialisez un `TabController` pour gérer les états des onglets.
- `length`: Nombre d'onglets.
- `vsync`: Synchronisation verticale pour les animations.

GESTION DES ÉVÉNEMENTS DE NAVIGATION

```
TabBar(  
  controller: _tabController,  
  onTap: (index) {  
    print("Selected Tab: $index");  
  },  
)
```

- `onTap`: Callback appelé lorsqu'un onglet est tapé.
- Utilisez `index` pour savoir quel onglet a été sélectionné.

STYLE ET PERSONNALISATION DU TABBAR

```
TabBar(  
  indicatorColor: Colors.red,  
  labelColor: Colors.green,  
  unselectedLabelColor: Colors.grey,  
)
```

- **indicatorColor**: Couleur de l'indicateur de l'onglet sélectionné.
- **labelColor**: Couleur du texte de l'onglet sélectionné.
- **unselectedLabelColor**: Couleur du texte des onglets non sélectionnés.

GRIDVIEW WIDGET

PRÉSENTATION DE GRIDVIEW

GridView est un widget dans Flutter qui permet d'afficher des éléments dans une grille bidimensionnelle. Il est utilisé pour créer des interfaces utilisateur flexibles et évolutives où les éléments sont automatiquement organisés selon la dimension de l'écran.

CRÉATION D'UN GRIDVIEW SIMPLE

Pour créer un GridView simple, vous pouvez utiliser le constructeur GridView.count :

```
GridView.count(  
    crossAxisCount: 2, // Nombre de colonnes  
    children: <Widget>[  
        Text('Element 1'),  
        Text('Element 2'),  
        Text('Element 3'),  
        Text('Element 4'),  
    ],  
)
```

GRIDVIEW.BUILDER

Le constructeur GridView.builder permet une création paresseuse d'éléments :

```
GridView.builder(  
    itemCount: 20,  
    gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 3),  
    itemBuilder: (context, index) {  
        return Text('Item $index');  
    },  
)
```

PROPRIÉTÉS DE GRIDVIEW (CROSSAXISCOUNT, MAINAXISSPACING, ETC.)

Propriété	Description
crossAxisCount	Définit le nombre de colonnes dans la grille.
mainAxisSpacing	L'espace entre les éléments dans l'axe principal.
crossAxisSpacing	L'espace entre les éléments dans l'axe transversal.
childAspectRatio	Le rapport entre la largeur et la hauteur de chaque enfant.

INTERACTION AVEC LES ÉLÉMENTS D'UN GRIDVIEW

Pour interagir avec les éléments d'un GridView, vous pouvez envelopper chaque élément dans un GestureDetector ou InkWell pour gérer les taps :

```
GridView.count(
  crossAxisCount: 2,
  children: List.generate(4, (index) {
    return InkWell(
      onTap: () => print('Tapped on item $index'),
      child: Container(
        padding: EdgeInsets.all(8),
        child: Text('Item $index'),
      ),
    );
  }),
)
```

ALERTDIALOG WIDGET

DÉFINITION DE ALERTDIALOG

AlertDialog est un widget dans Flutter qui affiche une boîte de dialogue modale, qui peut informer l'utilisateur ou recueillir des entrées de sa part. Il sert principalement à demander une confirmation ou à afficher des informations critiques.

UTILISATION DE ALERTDIALOG

Pour utiliser un AlertDialog, vous devez le placer dans une fonction asynchrone et l'afficher en utilisant `showDialog()`. Il est généralement utilisé pour des confirmations ou des alertes qui nécessitent une interaction de l'utilisateur.

STRUCTURE DE BASE D'UN ALERTDIALOG

```
AlertDialog(  
    title: Text('Titre'),  
    content: Text('Contenu de l\'alerte'),  
    actions: <Widget>[  
        FlatButton(  
            child: Text('Annuler'),  
            onPressed: () {  
                Navigator.of(context).pop();  
            },  
        ),  
        FlatButton(  
            child: Text('Accepter'),  
            onPressed: () {  
                // Actions à effectuer  
                Navigator.of(context).pop();  
            },  
        ),  
    ],
```

GESTION DES ACTIONS DANS ALERTDIALOG

Les actions dans un AlertDialog sont gérées par une liste de widgets, généralement des boutons, placés dans la propriété `actions`. Chaque bouton peut déclencher des fonctions incluant souvent la fermeture de la boîte de dialogue.

PERSONNALISATION DE L'APPARENCE DE ALERTDIALOG

Vous pouvez personnaliser l'apparence de l'AlertDialog en modifiant ses propriétés comme `backgroundColor`, `shape`, et `elevation`. Vous pouvez également utiliser `TextStyle` pour personnaliser les textes à l'intérieur de l'AlertDialog.

TOOLTIP WIDGET

INTRODUCTION AU TOOLTIP WIDGET

Un Tooltip est une étiquette d'information qui s'affiche lorsqu'un utilisateur passe le curseur sur un élément ou le touche pendant une durée prolongée. Dans Flutter, le **Tooltip** widget est utilisé pour fournir des explications supplémentaires aux utilisateurs sur les fonctionnalités des boutons ou d'autres éléments de l'interface utilisateur.

UTILISATION DE BASE DU TOOLTIP WIDGET

```
Tooltip(  
  message: 'Cliquez pour en savoir plus',  
  child: IconButton(  
    icon: Icon(Icons.info),  
    onPressed: () {  
      // Action du bouton  
    },  
  ),  
)
```

PERSONNALISATION DU TOOLTIP WIDGET

- **waitDuration**: Durée avant que le Tooltip apparaisse.
- **showDuration**: Durée d'affichage du Tooltip.
- **height**: Hauteur du Tooltip.
- **padding**: Marge intérieure du Tooltip.
- **decoration**: Décoration pour personnaliser l'apparence du Tooltip.

Exemple de personnalisation :

```
Tooltip(  
  message: 'Cliquez ici',  
  waitDuration: Duration(seconds: 1),  
  showDuration: Duration(seconds: 3),  
  decoration: BoxDecoration(  
    color: Colors.blue,  
    borderRadius: BorderRadius.circular(10),  
  ),  
  child: Icon(Icons.touch_app),  
)
```

EXEMPLES PRATIQUES AVEC TOOLTIP WIDGET

1. Tooltip sur un bouton:

```
Tooltip(  
  message: 'Envoyer un message',  
  child: ElevatedButton(  
    onPressed: () {},  
    child: Icon(Icons.send),  
  ),  
)
```

2. Tooltip sur une image:

```
Tooltip(  
  message: 'Logo Flutter',  
  child: Image.asset('assets/images/flutter_logo.png'),  
)
```