

TP3 : Application multifenêtres et gestion des évènements

Réalisé par : Aya NEJARI (Filière : ISITD)

Dans le cadre de ce TP sur Flutter, nous allons créer une application de gestion de liste de tâches (Todo List) qui nous permettra de mettre en pratique plusieurs concepts essentiels du développement d'applications mobiles.

- Structurer un projet Flutter de manière organisée en utilisant des modèles (models) pour représenter une structure de données (Classe dans POO)
- Concevoir l'interface utilisateur à l'aide de widgets, en utilisant notamment ListView, Card, Container, Flexible, Column, Row, TextField, Icon, ThemeData, AppBarTheme etc.
- Mettre en place des modal dialogs pour les interactions utilisateur, telles que l'ajout et la modification de tâches.
- Gérer et vérifier les entrées utilisateur, notamment pour la saisie de nouvelles tâches.
- Configurer et utiliser des thèmes d'application pour personnaliser l'apparence de l'application.
- Implémenter la gestion des événements pour permettre aux utilisateurs d'ajouter, de modifier ou de supprimer des tâches de la liste.

1. Création du projet

- Créer un nouveau dossier appelé TP3.
- A l'intérieur du dossier TP3, créer une application flutter en tapant la commande *flutter create*, appeler l'application *todolist_app*

```
PS>flutter create todolist_app
Creating project todolist_app...
█
```

- Accéder au dossier de votre application avec *cd*

In order to run your application, type:

```
$ cd todolist_app
$ flutter run
```

- Tester votre application avant de passer à l'étape suivante.
- Initialiser un dépôt git avec *git init*, et enregistrer votre avancement avec des commits.
A la fin de la séance faites un *git push* pour soumettre votre avancement dans le projet.
- Ajouter à votre dossier de projet un document contenant les réponses aux questions que vous rencontrerez durant le TP.

2. Création de la fonction main()

- Ouvrir le dossier main.dart se trouvant dans le dossier lib et y ajouter le contenu suivant :

```
import 'package:flutter/material.dart';
void main() {
  runApp(
    const MaterialApp(
      // home: ...,
    ),
  );
}
```

- Quel est le rôle de la fonction main (), de la fonction runApp ?

La fonction main() est point d'entrée de l'application tandis que la fonction runApp() est utilisée pour définir le widget racine de l'application et exécute le processus de l'interface utilisateur flutter.

- Que représente MaterialApp ?

MaterialApp est le premier widget de l'application flutter et représente sa configuration globale.

- Que représente home ?

Home représente la première interface utilisateur que l'utilisateur voit lors du lancement de l'application flutter.

3. Création de la widget Tasks

- Créer un nouveau fichier dart que vous appellerez *tasks.dart* et y coller le code suivant :

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

class Tasks extends StatefulWidget {
  const Tasks({super.key});

  @override
  State<Tasks> createState() {
    return _TasksState();
  }
}

class _TasksState extends State<Tasks> {
  @override
  Widget build(BuildContext context) {
```

```

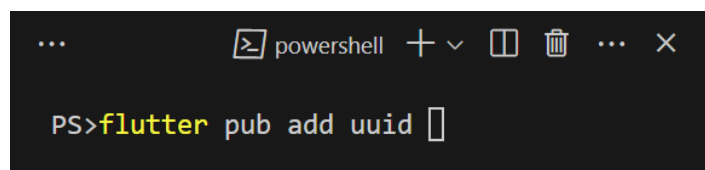
return Scaffold(
  body: Column(
    children: const [
      Text('The title'),
      Text('Tasks list...'),
    ],
  ),
);
}
}

```

- Tasks est une classe qui hérite de la classe StatefulWidget. Donner une justification ?
Tasks est une classe qui hérite de la classe StatefulWidget cela est justifié par le fait qu'on a créé un widget qui a un état qui change au fil du temps et donc il faut rafraîchir pour voir les modifications.
- A quoi sert la méthode createState() et qu'est-ce qu'elle retourne ?
La méthode createState() permet de créer un composant et lui donner un état et elle retourne un widget.
- Pourquoi faut-il créer une nouvelle classe appelée _TasksState ?
On crée une classe _TasksState pour séparer la logique de l'interface utilisateur de la logique d'état pour faciliter la maintenance et l'évolutivité du code.
- A quoi sert la méthode build() et qu'est-ce qu'elle retourne ?
La méthode build() permet de définir la présentation et elle retourne un widget.

4. Création du modèle (la classe task.dart)

- Dans le dossier lib, créer un nouveau dossier appelé *models*
- Depuis le terminal, taper la commande suivante:



```

... powershell + v [] [] ... X
PS>flutter pub add uuid []

```

- A quoi sert cette commande ?
Cette commande permet d'ajouter une dépendance spécifique c'est la bibliothèque « uuid ». « flutter pub » permet d'ajouter, de mettre à jour, de supprimer ou de gérer les dépendances du projet. « add » permet d'ajouter une nouvelle dépendance. « uuid » c'est le nom de la dépendance qu'on ajouté. "uuid" est une bibliothèque

Flutter qui permet de générer des identifiants uniques universels (UUID), conformément aux spécifications RFC 4122.

- Dans le dossier models, créer un nouveau fichier appelé task.dart
- Le modèle est responsable de la gestion des données de l'application. Il peut s'agir de données provenant d'une API, d'une base de données locale, ou de données stockées en mémoire. Les classes du modèle définissent la structure des données et éventuellement des méthodes pour les manipuler.
- Ajouter le code suivant à votre fichier appelé task.dart

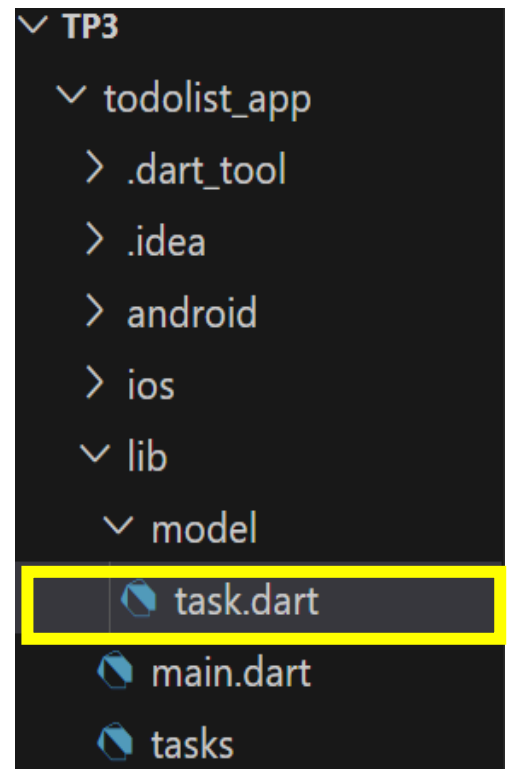
```
import 'package:uuid/uuid.dart';

const uuid = Uuid();

enum Category { personal, work, shopping, others }

class Task{
  Task({
    required this.title,
    required this.description,
    required this.date,
    required this.category,
  }) : id = uuid.v4();

  final String id;
  final String title;
  final String description;
  final DateTime date;
  final Category category;
}
```



: id = uuid.v4()); signifie que lorsqu'une instance de la classe Task est créée, un identifiant unique est généré et assigné au champ id.

- Pourquoi les propriétés de la classe Task sont considérés comme final ? Quel est l'objectif ?

Les propriétés de la classe Task sont considérés comme final pour indiquer qu'elles sont immuables une fois qu'elles sont définies lors de la création d'une instance de la classe. Cela signifie que ces propriétés ne peuvent pas être modifiées après avoir été initialisées dans le constructeur de la classe. L'objectif de cette conception est d'assurer l'immutabilité des données.

- Revenir au fichier `main.dart` et changer son contenu comme suit :

```
import 'package:flutter/material.dart';  
  
import 'package:todoist_app/tasks';  
  
void main() {  
  runApp(  
    const MaterialApp(  
      home: Tasks(),  
    ),  
  );  
}
```

- Ouvrir le fichier `tasks.dart`

Pour le moment, nous souhaitons afficher des données statiques dans notre application mobile.
Pour cela, nous allons créer une liste de tâches que nous appellerons `_resgistrdTasks`

```
class _TasksState extends State<Tasks> {  
  final List<Task> _registeredTasks = [  
    Task(  
      title: 'Apprendre Flutter',  
      description: 'Suivre le cours pour apprendre de nouvelles compétences',  
      date: DateTime.now(),  
      category: Category.work,  
    ),  
    Task(  
      title: 'Faire les courses',  
      description: 'Acheter des provisions pour la semaine',  
      date: DateTime.now().subtract(Duration(days: 1)),  
      category: Category.shopping,  
    ),  
    Task(  
      title: 'Rediger un CR',  
      description: '',  
      date: DateTime.now().subtract(Duration(days: 2)),  
      category: Category.personal,  
    ),  
    // Add more tasks with descriptions as needed  
  ];  
}
```

5. Création de la widget `TasksList`

- Maintenant que vous avez ajouté des données fictives pour démarrer, il est important de les afficher dans un widget de type liste.
- A l'intérieur du dossier `lib` créer un nouveau widget que vous appellerez `tasks_list.dart`

- Expliquer pourquoi ce Widget est de type StatelessWidget ? Lire le code suivant et déduire son rôle

Le widget `TasksList` est de type `StatelessWidget` parce qu'il n'a pas besoin de gérer un état interne mutable. Il est conçu pour afficher la liste de tâches reçue en tant que propriété (qui est un état externe) et n'a pas besoin de modifier cet état. Un widget de type `StatelessWidget` est approprié pour les cas où l'interface utilisateur du widget est purement basée sur les propriétés entrantes et ne nécessite pas de mise à jour d'état interne.

Le rôle du widget `TasksList` est d'afficher la liste des tâches (qui est une liste de `Task`) dans une liste déroulante.

```
import 'package:flutter/material.dart';
import 'package:todolist_app/model/task.dart';

class TasksList extends StatelessWidget {
  const TasksList({
    super.key,
    required this.tasks,
  });

  final List<Task> tasks;

  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: tasks.length,
      itemBuilder: (ctx, index) => Text(tasks[index].title),
    );
  }
}
```

`ListView` est un widget dans Flutter qui permettra d'afficher une liste déroulante contenant des tâches à exécuter. Pour gérer efficacement de grandes listes de données, on utilise souvent la méthode `builder`. Cette méthode génère dynamiquement les éléments de la liste à mesure qu'ils sont affichés à l'écran, ce qui améliore les performances de l'application et réduit la consommation de mémoire.

`itemBuilder: (ctx, index) => Text(tasks[index].title),`

Cette ligne de code indique au `ListView.builder` de créer un widget `Text` pour chaque élément de la liste `tasks` et y afficher le titre de la tâche.

Le paramètre ctx (ou BuildContext) est une variable automatiquement reconnue par Flutter. La valeur ajoutée de BuildContext réside dans le fait qu'il est généralement utilisé dans des situations où vous avez besoin d'accéder à des informations spécifiques à la construction des widgets, telles que les thèmes, la localisation, la taille de l'écran, et d'autres données contextuelles.

Vous pouvez également réécrire la même ligne de cette façon :

```
itemBuilder: (_, index) => Text(tasks[index].title),
```

Dans cet exemple, _ est une convention courante pour indiquer que vous ignorez intentionnellement le paramètre.

- A quoi sert la flèche => ? Comment appelle-t-on ce genre de fonction en dart ?

La flèche => est utilisée en Dart pour définir des fonctions anonymes ou des expressions lambda (lambda functions). Ce type de fonction est également appelé une fonction fléchée (arrow function) en Dart. Les fonctions fléchées sont couramment utilisées pour créer des fonctions simples et courtes de manière concise.

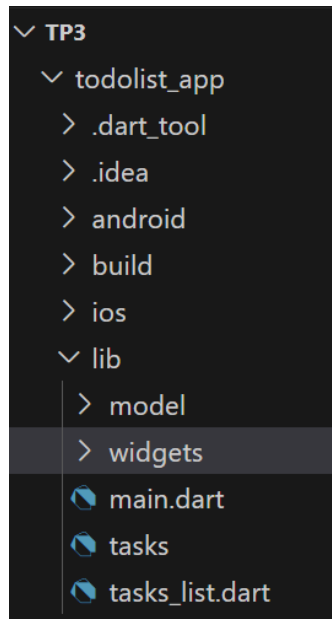
- Réécrire le même code sans => ? Que constatez-vous ?

Sans utiliser la flèche =>, on peut définir la même fonction en utilisant la syntaxe complète de la fonction anonyme, on écrit le code comme ça : `itemBuilder:`
`(BuildContext ctx, int index) {`
 `return Text(tasks[index].title);`
`}`

On constate qu'on a déclaré explicitement le type des paramètres (BuildContext ctx et int index) et on a utilisé les accolades {} pour définir le corps de la fonction.

Une meilleure organisation des Widgets

- Remarquez que le nombre de widget devient important. A ce stade, il est recommandé d'ajouter un dossier qui va contenir tous vos widgets. L'intérêt est de faire la distinction entre les structures de données manipulées (Dossier Models) et les Widgets qui sont des éléments graphiques qui se trouvent dans le dossier Widgets.



6. Création de la widget TaskItem.

- Dans le dossier widgets, ajouter un nouveau fichier appelé task_item.dart.
- A l'intérieur de ce fichier, créer une nouvelle classe (Widget) appelée TaskItem. Générer le constructeur + la méthode build.

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

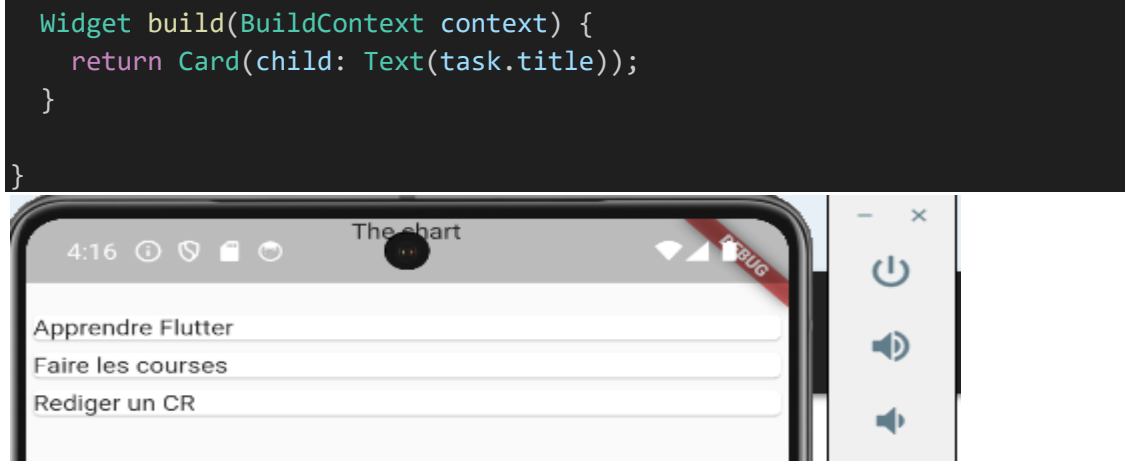
class TaskItem extends StatelessWidget{
  const TaskItem({super.key});

  @override
  Widget build(BuildContext context) {
    // TODO: implement build
    throw UnimplementedError();
  }
}
```

- Modifier le code de la classe TaskItem pour ajouter une nouvelle propriété appelée task de type Task, modifier le constructeur et la méthode build comme suit.

```
import 'package:flutter/material.dart';
import 'package:todolist_app/model/task.dart';

class TaskItem extends StatelessWidget{
  const TaskItem(this.task, {super.key});
  final Task task;
  @override
```

- Exécuter et vérifier que le résultat.
- A quoi sert la widget Card ?

Le widget Card permet d'afficher le titre de la tâche, ce qui rend chaque élément de la liste de tâches plus attrayant et facile à distinguer. Le résultat est une liste de tâches où chaque tâche est encapsulée dans un widget Card, avec le titre de la tâche affiché à l'intérieur de celui-ci.

7. Ajouter le thème et l'AppBar

- Revenir au fichier appelé main.dart
- Modifier ce widget en se basant sur le code suivant :

```
void main() {  
  runApp(  
    MaterialApp(  
      theme: ThemeData(  
        appBarTheme: const AppBarTheme(elevation: 0),  
        useMaterial3: true,  
      ),  
      home: const Tasks(),  
    ),  
  );  
}
```

- A quoi sert le widget ThemeData et AppBarTheme ?

ThemeData est un widget qui permet de définir un thème global pour l'application. On a utilisé theme pour spécifier un thème global. Le thème permet de personnaliser l'apparence de l'application, y compris les couleurs, les polices, les élévations, etc. On a utilisé AppBarTheme pour personnaliser l'apparence des barres d'applications. AppBarTheme est un sous-ensemble du thème qui permet de personnaliser les barres d'applications (app bars) dans l'application. Dans ce code, on a défini elevation: 0 dans

AppBarTheme, ce qui signifie qu'on a supprimé l'ombre (élévation) par défaut des barres d'applications. Cela donne l'apparence d'une barre d'application plate sans ombre.

- Revenir à votre Widget Scaffold et ajouter l'AppBar comme le montre le code ci-dessous

```
return Scaffold(  
  appBar: AppBar(  
    title: const Text('Flutter ToDoList'),  
    actions: [  
      IconButton(  
        onPressed: () {},  
        icon: const Icon(Icons.add),  
      ),  
    ],  
  ),  
  body: Column(  
    ...
```

- Remarquez que vous avez ajouté une icône + permettant d'ajouter une nouvelle tâche à votre list.

- Quel est le rôle de la méthode onPressed() ? pourquoi nous avons ajouté {}

La méthode onPressed() est utilisée pour spécifier ce qui se passe lorsque l'utilisateur appuie sur le bouton "Ajouter" (représenté par l'icône "+" dans la barre d'applications). La syntaxe onPressed: () {} indique qu'une fonction de rappel vide (lambda function) doit être exécutée lorsqu'on appuie sur le bouton. Dans cet exemple, on a laissé la fonction vide (utilisation de {}) pour le moment, ce qui signifie que rien ne se produira lorsqu'on appuie sur le bouton "Ajouter". Cependant, on peut ajouter le code approprié à l'intérieur des accolades {} pour définir le comportement souhaité lorsque le bouton est pressé.

- Modifier le code associé à l'icône comme suite :

```
IconButton(  
  onPressed: _openAddTaskOverlay,  
  icon: const Icon(Icons.add),  
),
```

- Juste avant la fonction build() ajouter la définition de la méthode _openAddTaskOverlay()

```
void _openAddTaskOverlay() {
  showModalBottomSheet(
    context: context,
    builder: (ctx) => const Text('Exemple de fenêtre'),
  );
}
```

- A quoi sert showModalBottomSheet?

showModalBottomSheet est utilisé pour créer une expérience utilisateur où l'utilisateur peut ajouter une nouvelle tâche ou effectuer d'autres actions spécifiques à partir d'une feuille de bas de page modale, tout en masquant temporairement le reste de l'application.

- Pour le moment lorsque vous appuyez sur l'icône plus, une fenêtre s'affiche avec un simple texte qui s'affiche. A ce stade, nous allons personnaliser cette fenêtre pour afficher un widget sous forme de formulaire nous permettant la saisie de données ;

8. Création d'un Widget pour la saisie des données

- Dans le dossier Widget, créer un nouveau fichier que vous appellerez new_task.dart
- Ajouter le code suivant au fichier new_task.dart

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

class NewTask extends StatefulWidget{
  const NewTask({super.key});
  @override
  State<NewTask> createState() {
    return _NewTaskState();
  }
}

class _NewTaskState extends State<NewTask>{
  @override
  Widget build(BuildContext context) {
    return const Text('Hello');
  }
}
```

- Expliquer pourquoi la classe NewTask est de type StatefulWidget ?

La classe NewTask est de type StatefulWidget parce qu'elle est destinée à contenir des états mutables, et elle peut être reconstruite avec de nouveaux états à mesure que les données changent.

- Modifier le code précédent de façon à ajouter un petit formulaire avec une zone de saisie :

```
return const Padding(
  padding: EdgeInsets.all(16),
  child: Column(
    children: [
      TextField(
        maxLength: 50,
        decoration: InputDecoration(
          label: Text('Title'),
        ),
      ),
    ],
  ),
);
```

- Ajouter un bouton permettant d'enregistrer la nouvelle tâche :

```
children: [
  TextField(
    .....,
  ),
  Row(
    children: [
      ElevatedButton(
        onPressed: () {
          print( 'Vous avez appuyé sur Save');
        },
        child: const Text('Save Task'),
      ),
    ],
  ),
],
```

- Revenir au fichier tasks.dart, et modifier le code comme suit :

```
void _openAddTaskOverlay() {
  showModalBottomSheet(
    context: context,
    builder: (ctx) => const NewTask(),
  );
}
```

- Tester votre application

9. Récupérer la saisie de l'utilisateur

Methode 1 : à l'aide d'une fonction

- Ouvrir le fichier new_task.dart
- Se positionner sur la classe _NewTaskState

- Ajouter le code suivant, juste avant la méthode build.

```
class _NewTaskState extends State<NewTask>{

  var _enteredTitle = '';

  void _saveTitleInput(String inputValue) {
    _enteredTitle = inputValue;
  }
}
```

- La fonction _saveTitleInput est une fonction de rappel (callback) qui vous permet de réagir aux modifications du texte à mesure qu'elles se produisent. Au fur et à mesure que l'utilisateur tape au clavier, ce qu'il écrit est directement récupéré par cette fonction.
- onChanged est un événement associé à un widget TextField. L'événement onChanged est déclenché chaque fois que le texte dans le champ de texte change. Changer le code associé à votre zone de texte pour prendre en considération l'événement onChanged.

```
children: [
  TextField(
    onChanged: _saveTitleInput,
    maxLength: 50,
    decoration: const InputDecoration(
      label: Text('Task title'),
    ),
  ),
],
```

- Modifier le code associé au bouton «Enregistrer » comme suit :

```
ElevatedButton(
  onPressed: () {
    print(_enteredTitle);
  },
  child: const Text('Enregistrer'),
),
```

- Exécuter le code, ouvrir la fenêtre de débogage : Menu affichage -> Console de débogage.
- Ecrire un exemple de tâche et constater l'affichage.

Méthode 2 : à l'aide d'un contrôleur

- Toujours dans fichier new_task.dart, ajouter le code suivant :

```
class _NewTaskState extends State<NewTask>{
  final _titleController = TextEditingController();
}
```

```
@override
void dispose() {
  _titleController.dispose();
  super.dispose();
}
```

- A quoi sert un contrôleur ? Quel est le rôle de la méthode dispose()

Un contrôleur est un objet utilisé pour gérer la saisie de texte ou d'autres informations dans un champ de texte (comme un TextField ou un TextFormField) dans Flutter. Il permet de lier le champ de texte à un objet qui peut être utilisé pour accéder, lire et mettre à jour le texte saisi par l'utilisateur.

La méthode dispose() est une méthode de désallocation dans Flutter. Elle est appelée lorsque l'objet de l'état d'un widget est supprimé ou lorsque le widget lui-même est retiré de l'arbre des widgets. La méthode dispose() est utilisée pour libérer les ressources associées au contrôleur, en particulier pour éviter les fuites de mémoire.

- Changer le code associé à l'évènement onChanged() précédemment et le remplacer par controller.

```
children: [
  TextField(
    controller: _titleController,
    maxLength: 50,
    decoration: const InputDecoration(
      label: Text('Task title'),
    ),
  ),
],
```

Changer le code associé au bouton enregistrer comme suit :

```
ElevatedButton(
  onPressed: () {
    print(_titleController.text);
  },
  child: const Text('Enregistrer'),
),
```

A votre avis quelle est la meilleure méthode pour récupérer la saisie de l'utilisateur ?

Personnellement, je trouve que la meilleure méthode pour récupérer la saisie de l'utilisateur est la deuxième méthode à l'aide du contrôleur car le contrôleur gère automatiquement la valeur saisie dans le champ de texte, stocke et met à jour la saisie de l'utilisateur. Il est aussi utile pour des formulaires et des entrées de données où on a besoin de gérer la valeur du champ de manière globale.

10. Contrôler la saisie de l'utilisateur

Maintenant, au lieu d'afficher uniquement ce qui a été saisi, il serait judicieux de récupérer les données et vérifier leur conformité

- Modifier onPressed pour appeler la méthode _submitTaskData comme suit

```
onPressed: _submitTaskData,
```

- Créer la méthode _submitTaskData() pour vérifier la saisie de l'utilisateur

```
void _submitTaskData() {  
  
  if (_titleController.text.trim().isEmpty) {  
    showDialog(  
      context: context,  
      builder: (ctx) => AlertDialog(  
        title: const Text('Erreur'),  
        content: const Text(  
          'Merci de saisir le titre de la tâche à ajouter dans la liste'),  
        actions: [  
          TextButton(  
            onPressed: () {  
              Navigator.pop(ctx);  
            },  
            child: const Text('Okay'),  
          ),  
        ],  
      ),  
    );  
    return;  
  }  
}
```