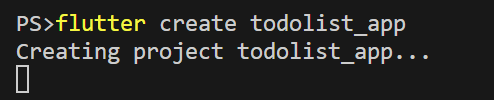
**TP3 : Application multifenêtres et gestion des évènements**

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



* Accéder au dossier de votre application avec cd

A black screen with white text

Description automatically generated

* 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.

1. **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() permet d’initialiser l’application et la fonction runApp() permet d’exécuter le projet.

* Que représente MaterialApp ?

Est un widget qui permet de définir le thème de l’application.

* Que représente home ?

Définit l’écran initial de l’application.

1. **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 ?

Car il y a des composantes qui changent de l’état.

* A quoi sert la méthode **createState()** et qu’est-ce qu’elle retourne ?

Elle est utilisée pour instancier l’objet ‘State’ qui gère l’etat d’un widget .elle retourne un widget.

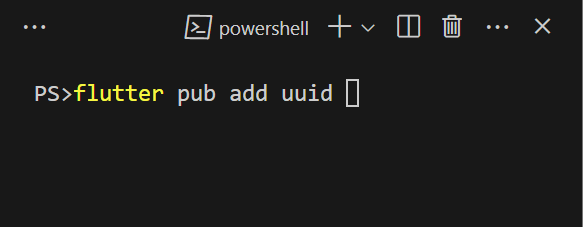
* Pourquoi faut-il créer une nouvelle classe appelée **\_TasksState** ?
* A quoi sert la méthode **build()** et qu’est- ce qu’elle retourne ?

Utilisée pour construire l’interface utilisateur d’un widget par rapport à son état.

elle retourne un widget.

1. **Création du modèle (la classe task.dart)**

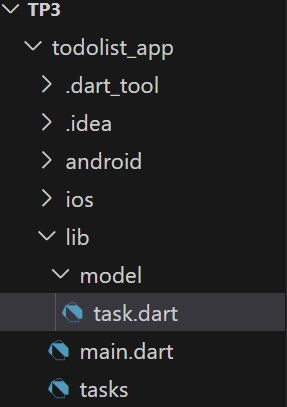
* Dans le dossier lib, créer un nouveau dossier appelé ***models***
* Depuis le terminal, taper la commande suivante:



* A quoi sert cette commande ?

Elle permet d’installer certaines indépendances.

* 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 ?*

*Ces propriétés sont des constantes, elles ne doivent pas être modifier .*

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

import 'package:flutter/material.dart';

import 'package:todolist\_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 \_resgistredTasks

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

];

1. **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 StatelessWiget ? Lire le code suivant et déduire son rôle.

**‘TasksList’** prend une liste de tâches en entrée. Il affiche ensuite ces tâches sous forme de texte dans un ‘**ListView’**.

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 c’est utilisée pour définir une fonction anonyme.

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

1. **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.

A screenshot of a computer

Description automatically generated

1. **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 apelée 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

  Widget build(BuildContext context) {

    return Card(child: Text(task.title));

  }

A screenshot of a computer

Description automatically generated}

* Exécuter et vérifier que le résultat.
* A quoi sert la widget Card ?
* widget **Card** est utilisé pour créer une présentation visuelle agréable de la tâche en enveloppant le texte dans une carte. Cela ajoute une certaine structure et un style à l'affichage du titre de la tâche, en utilisant les paramètres par défaut de la carte pour l'arrière-plan, les bords et l'ombre.

1. **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 utilisé pour définir un thème global pour l'application, tandis que **AppBarTheme** est une sous-configuration de ce thème qui permet de personnaliser l'apparence des barres d'applications (app bar)
* 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 ajouter une icône + permettant d’ajouter une nouvelle tâche à votre list.
* Quel est le rôle de la méthode onPressed() ? pourquoi nous avons ajouté {}
* onPressed() est une fonction qui est exécutée lorsque l'événement associé se produit. Dans ce cas, elle est associée à un **IconButton**, ce qui signifie que lorsque l'utilisateur appuie sur cet icône, la fonction spécifiée dans **onPressed** sera exécutée.
* il y a une fonction anonyme vide **() {}** qui signifie qu'aucune action particulière n'est définie pour le moment. Cela indique que lorsqu'on appuie sur le bouton, rien ne se passe pour le moment.
* 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 et afficher une fenêtre modale qui s'affiche en bas de l'écran. Cette fenêtre modale peut contenir n'importe quel widget, mais dans ce cas, il s'agit d'un simple texte.
* 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 ;

1. **Création d’un Widget pour la saisie des données**

* Dans le dossier Widget, créer un nouveau fichier que vous appelerez 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 définie comme un **StatefulWidget** car elle peut avoir un état interne qui peut changer au fil du temps, et elle doit être capable de réagir à des interactions utilisateur ou d'autres événements.
* 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

1. **Récupérer la saisie de l’utilisateur**

**Methode 1 : à l’aide d’une focntion**

* 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’évènement 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 (**TextEditingController** ) est un objet qui permet de gérer le texte saisi par l'utilisateur dans un champ de texte (**TextField**) .
* la méthode **dispose()** est appelée lorsqu'un objet est retiré de l'arborescence de widgets et sert à libérer les ressources associées 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 saisir de l’utilisateur ?

1. **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;

    }

   }