

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*

```
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 la fonction principale du programme, runApp est le moteur qui exécute l'application.
- Que représente MaterialApp ?
 - MaterialApp fournit la structure des applications Flutter.
- Que représente home ?
 - home représente la première page et la principale qui sera affichée.

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...'),
        ],
      ),
    );
  }
}
```

```

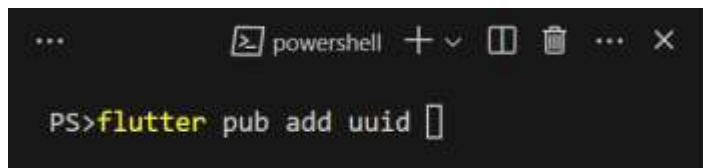
    ],
  ),
);
}
}
}

```

- Tasks est une classe qui hérite de la classe StatefulWidget. Donner une justification ?
 - La classe Tasks hérite de la classe StatefulWidget pour permettre aux instances de gérer l'état interne.
- A quoi sert la méthode createState() et qu'est-ce qu'elle retourne ?
 - La méthode createState() permet de retourner un task créé.
- Pourquoi faut-il créer une nouvelle classe appelée _TasksState ?
 - La gestion de ce que retourne la fonction createState()
- A quoi sert la méthode build() et qu'est-ce qu'elle retourne ?
 - La méthode build() est utilisée pour construire et retourner l'interface utilisateur d'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 + - [ ] [ ] ... X
PS>flutter pub add uuid

```

- A quoi sert cette commande ?

```

Resolving dependencies...
  async 2.10.0 (2.11.0 available)
  characters 1.2.1 (1.3.0 available)
  collection 1.17.0 (1.18.0 available)
+ crypto 3.0.3
  js 0.6.5 (0.6.7 available)
  lints 2.0.1 (2.1.1 available)
  matcher 0.12.13 (0.12.16 available)
  material_color_utilities 0.2.0 (0.8.0 available)
  meta 1.8.0 (1.11.0 available)
  path 1.8.2 (1.8.3 available)
  source_span 1.9.1 (1.10.0 available)
+ sprintf 7.0.0
  stack_trace 1.11.0 (1.11.1 available)
  stream_channel 2.1.1 (2.1.2 available)
  test_api 0.4.16 (0.6.1 available)
+ typed_data 1.3.2
+ uuid 4.1.0
Changed 4 dependencies!

```

- Cette commande sert à ajouter la dépendance uuid au projet.

- 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

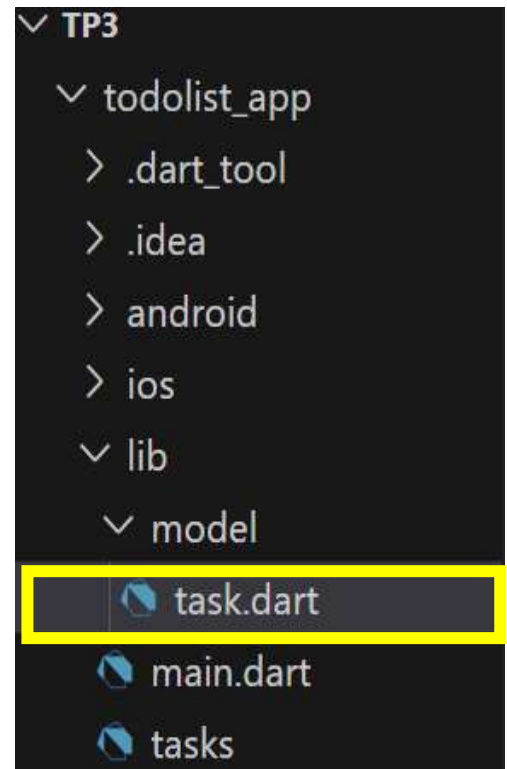
```
import 'package:uuid/uuid.dart';

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?
 - Ils sont considérés comme final afin de ne jamais changer leurs valeurs.
- 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 `_resgistedTasks`

```
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
 - Son rôle est seulement d'afficher une liste de tâches sous forme d'une ListView où chaque tâche est affichée sous forme de texte.

```
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 ?

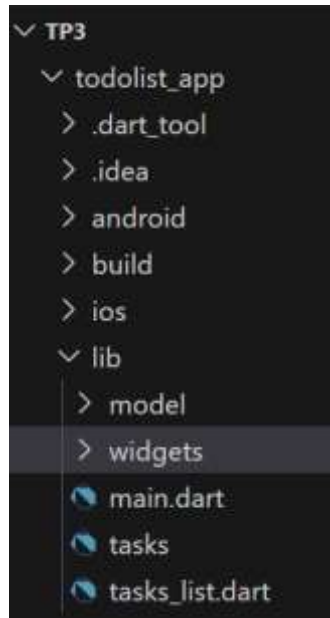
- La flèche => est utilisée pour définir une fonction anonyme en Dart.

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

- itemBuilder: (BuildContext ctx, int index) {
 return Text(tasks[index].title);
 } Cette méthode affiche le même résultat que la fonction anonyme.

6. 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.



7. 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:todoist_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));
  }
}
```



- Exécuter et vérifier que le résultat.
- A quoi sert la widget Card ?
 - La widget Card est utilisé pour afficher le titre de la tâche (`task.title`) avec un effet de carte.

8. 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 définit la configuration du thème visuel global pour un MaterialApp, AppBarTheme personnaliser les couleurs, la luminosité
- 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 {}
 - Si `onPressed` est défini, ce rappel sera appelé lorsque l'utilisateur appuie sur bouton +, Les accolades {} indiquent un bloc de code vide, permettant de définir une action personnalisée à l'intérieur.
- 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 afficher une fenêtre modale en bas de l'écran, pour afficher des informations sans naviguer vers une nouvelle page

- 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 ;

9. 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 ?
 - NewTask est destiné à représenter un widget qui peut avoir un état interne modifié
- 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

10. 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 utilisé pour gérer les données d'entrée d'un widget, en permettant d'accéder et de modifier les données. La méthode dispose() est utilisée pour libérer les ressources et effectuer la maintenance finale d'un widget

- 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 ?

- La meilleure méthode pour récupérer la saisie de l'utilisateur est d'utiliser un contrôleur .

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;
}
}

```