Flutter TD3

Ajout d'un widget pour le choix du thème

Ajoutons maintenant à notre BottomNavigationBar un nouvel Icon qui va permettre à l'utilisateur de choisir le thème de l'application via le nouveau widget suivant qui utilise le package **settings_ui** disponible ici.

```
class EcranSettings extends StatefulWidget{
  @override
  State<EcranSettings> createState() => _EcranSettingsState();
}

class _EcranSettingsState extends State<EcranSettings> {
  bool _dark =true;
  @override
  Widget build(BuildContext context) {
    return Center(
      child: SettingsList(
        darkTheme: SettingsThemeData(
          settingsListBackground: MyTheme.dark().scaffoldBackgroundColor,
          settingsSectionBackground:
MyTheme.dark().scaffoldBackgroundColor
        ),
        lightTheme: SettingsThemeData(
          settingsListBackground: MyTheme.light().scaffoldBackgroundColor,
          settingsSectionBackground:
MyTheme.light().scaffoldBackgroundColor
        ),
        sections: [
          SettingsSection(
              title: const Text('Theme'),
              tiles: [
                SettingsTile.switchTile(
                    initialValue: _dark,
                    onToggle: _onToggle,
                    title: const Text('Dark mode'),
                leading: const Icon(Icons.invert_colors),)
              ])
        ],
      ),
    );
  }

  _onToggle(bool value) {
    debugPrint('value $value');
    setState(() {
      _dark = !_dark;
    });
  }
}
```

Persistance des données

Dans une application, il est souvent nécessaire de sauvegarder d'une utilisation à l'autre afin que l'utilisateur puisse retrouver certaines informations comme par exemple le fait d'être bien identifié, ... ou dans notre cas le thème choisi. Pour des données "simples", il existe les **SharedPreferences** qui permettent de sauvegarder des informations sous forme de clé/valeur dans un fichier. Pour cela, nous avons besoin d'un package disponible ici. Afin d'organiser au mieux notre code, créons une classe chargée de gérer l'accès aux données (un **repository**).

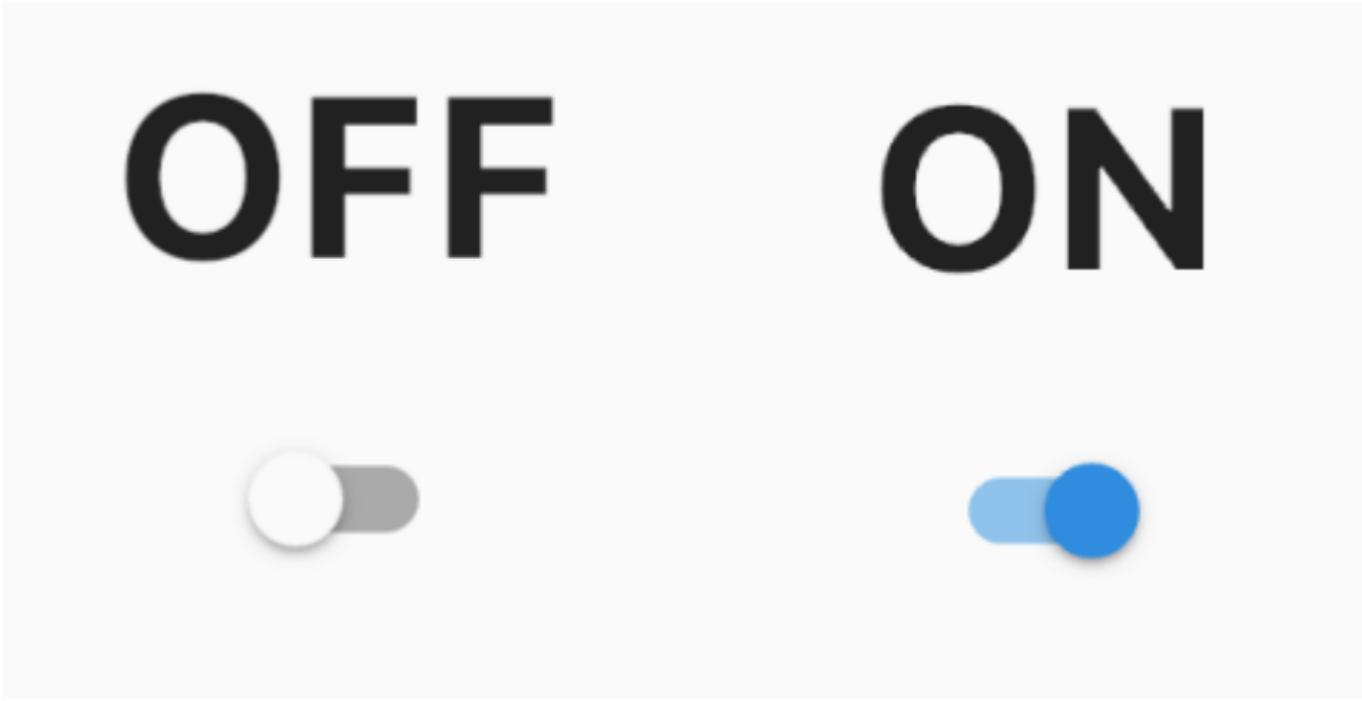
```
class  SettingRepository{
  static const THEME_KEY = "darkMode";

  saveSettings(bool value) async {
    SharedPreferences sharedPreferences = await
SharedPreferences.getInstance();
    sharedPreferences.setBool(THEME_KEY, value);
  }

  Future<bool> getSettings() async {
    SharedPreferences sharedPreferences = await
SharedPreferences.getInstance();
    return sharedPreferences.getBool(THEME_KEY) ?? false;
  }
}
```

State Management

En Flutter, un état représente les données dynamiques qui affectent ce qui est affiché à l'écran. Pour rappel, à chaque widget dynamique (StatefullWidget) est associé un état et il est possible de demander au widget de se "regénérer" lors d'un changement de son état.

Dans une application un peu complexe, une interface utilisateur peut comporter un nombre important de widgets dynamiques dont l'état de certains peut par exemple de l'état d'autres widgets.

Exemple:



C'est le cas dans notre application car le thème qui est défini à la racine de l'arbre des widgets dépend du wiget utilisé pour le choix du thème.

Plus généralement, on doit en permanance être capable de faire circuler des données dans l'arbre des widgets et donc gérer correctement les états de tous les widgets de façon correcte et efficace tout en concervant un code lisible, testable et maintenable peut rapidement devenir assez complexe.

C'est pourquoi Flutter propose un ensemble de solutions (environ une trentaine) appelées de façon générique **state management** ayant chacune des avantages et des incovéniants selon certains critères comme par exemple la complexité et la quantité de code nécessaire pour la mise en oeuvre, la mise en place de tests, ...

Toutes ces approches peuvent être classées dans trois grandes catégories:

- celles s'appuyant sur les classes natives du sdk.
- celles basées sur le paradigm de programmation **reactive**.
- celles utilisant les concepts de **programmation fonctionnelle**.

La documentation Flutter propose une sélection disponible ici.

Pour notre application, nous allons utiliser le package **provider** qui est recommandé par Flutter car il est relativement simple à comprendre, à mettre en oeuvre et assez efficace pour gérer l'état d'une application. Son implémentation s'appuie sur les **InheritedWidgets** du SDK et sa documentation est ici.

ViewModel

Les **ViewModels** sont des classes permettant de fournir des données et donc les états à l'interface utilisateur et contiennent la logique métier associée. Un ViewModel hérite de **ChangeNotifier** qui est une API permettant d'envoyer des notifications de "changement". Voici le ViewModel que l'on va utiliser pour obtenir/modifier le thème de notre application.

```
class SettingViewModel extends ChangeNotifier {
  late bool _isDark;
```

```
  late SettingRepository _settingRepository;
  bool get isDark => _isDark;

  SettingViewModel() {
    _isDark = false;
    _settingRepository = SettingRepository();
    getSettings();
  }

//Switching the themes
  set isDark(bool value) {
    _isDark = value;
    _settingRepository.saveSettings(value);
    notifyListeners();
  }

  getSettings() async {
    _isDark = await _settingRepository.getSettings();
    notifyListeners();
  }
}
```

La méthode **notifyListeners** permet de déclencher la méthode **build** des widgets qui l'appellent.

Utilisation de provider dans l'application

Après avoir créé les ViewModels, ils doivent être placés en haut de l'arborescence des widgets à l'aide du package provider. Ce package dispose d'un widget **MultiProvider** qui permet de fournir un tableau de ChangeNotifierProviders, chacun étant responsable de créer puis transmettre dans l'arborescence un ViewModel. La propriété create prend une fonction qui doit renvoyer une instance de ChangeNotifier (un ViewModel). Une bonne pratique consiste à faire quelques réglages initiaux avant de renvoyer l'instance. Par exemple, nous pouvons appeler une fonction pour récupérer notre thème, ce qui le rendra disponible avant même que les widgets ne soient dessinés sur l'écran. Pour le moment, nous disposons uniquement d'un seul provider donc nous n'utilisonserons pas encore Multiprovider.

```
class MyTD2 extends StatelessWidget{
  @override
  Widget build(BuildContext context){
    //creation d'un provider
    return ChangeNotifierProvider(
        create: (_){
          SettingViewModel settingViewModel = SettingViewModel();
          //getSettings est deja appelee dans le constructeur
          return settingViewModel;
        },
      child: Consumer<SettingViewModel>(
        builder: (context,SettingViewModel notifier,child){
          return MaterialApp(
              theme: notifier.isDark ? MyTheme.dark():MyTheme.light(),
              title: 'TD2',
```

```
                home: Home()
            );
          },
        ),
      );
    }
}
```

Dans cet exemple, le viewModel est utilisé dans le widget où il a été initialisé. Dans ce cas, il faut obligatoirement utiliser **Consumer**.

Le widget **EcranSettings** utilise lui aussi notre viewmodel.

```
class _EcranSettingsState extends State<EcranSettings> {
  @override
  Widget build(BuildContext context) {
          return Center(
            child: SettingsList(
              darkTheme: SettingsThemeData(
                  settingsListBackground:
MyTheme.dark().scaffoldBackgroundColor,
                  settingsSectionBackground:
MyTheme.dark().scaffoldBackgroundColor
              ),
              lightTheme: SettingsThemeData(
                  settingsListBackground:
MyTheme.light().scaffoldBackgroundColor,
                  settingsSectionBackground:
MyTheme.light().scaffoldBackgroundColor
              ),
              sections: [
                SettingsSection(
                    title: const Text('Theme'),
                    tiles: [
                      SettingsTile.switchTile(
                        initialValue: context.watch<SettingViewModel>
().isDark, //Provider.of<SettingViewModel>(context).isDark,
                        onToggle: (bool value)
{context.read<SettingViewModel>
().isDark=value;},//Provider.of<SettingViewModel>
(context,listen:false).isDark=value;},
                        title: const Text('Dark mode'),
                        leading: const Icon(Icons.invert_colors),)
                    ])
              ],
            ),
          );
  }

}
```

Ici, nous avons utilisé context.watch<SettingViewModel>() (resp. Provider.of<SettingViewModel>(context)) pour accéder à notre viewModel. La syntaxe est plus simple que celle de **consumer** mais ce dernier permet entre autre une gestion plus fine des widgets à redessiner ou pas lors de la mise à jour de l'interface quand les données changent dans le viewmodel.

context.read<SettingViewModel>() (resp. Provider.of<SettingViewModel>(context,listen:false)) permettent également d'accéder au viewModel mais dans ce cas, pour faire simple, le viewModel ne declenche pas de mise à jour.

Modification du modèle Task

Afin de poursuivre le développement de notre application, ajoutons un factory à notre classe **Task**.

```
factory Task.newTask(){
    nb++; //attribut static de la classe.
    return Task(id: nb, title: 'title $nb', tags: ['tags $nb'], nbhours:
nb, difficulty: nb%5, description: 'description $nb');
  }
```

Ajout du viewModel

Dans notre cas, on se dispense de créer un repository pour les Tasks car notre application est très simple. Voici le viewModel que nous allons utiliser:

```
class TaskViewModel extends ChangeNotifier{
  late List<Task> liste;

  TaskViewModel(){
    liste=[];
  }

  void addTask(Task task){
    liste.add(task);
    notifyListeners();
  }

  void generateTasks(){
    liste = Task.generateTask(50);
    notifyListeners();
  }
}
```

Ajout du viewModel à notre application

Ajoutons maintenant notre viewModel grâce au widget **Multiprovider** du package provider.

```
class MyTD2 extends StatelessWidget{
  @override
  Widget build(BuildContext context){
    return MultiProvider(
        providers: [
          ChangeNotifierProvider(
            create: (_){
              SettingViewModel settingViewModel = SettingViewModel();
              //getSettings est deja appelee dans le constructeur
              return settingViewModel;
          }),
          ChangeNotifierProvider(
              create:(_){
                TaskViewModel taskViewModel = TaskViewModel();
                taskViewModel.generateTasks();
                return taskViewModel;
              } )
        ],
      child: Consumer<SettingViewModel>(
        builder: (context,SettingViewModel notifier,child){
          return MaterialApp(
              theme: notifier.isDark ? MyTheme.dark():MyTheme.light(),
              title: 'TD2',
              home: Home()
          );
        },
      ),
    );
  }
}
```

Utilisation de notre ViewModel

Notre viewModel fournit maintenant la liste des Tasks.

```
class Ecran1 extends StatelessWidget{
  late List<Task> tasks; //= Task.generateTask(50);
  String tags='';
  @override
  Widget build(BuildContext context) {
    tasks = context.watch<TaskViewModel>().liste;
    return ListView.builder(
        ...
    };
}
```

Navigation vers un widget pour ajouter une Task

Dans notre **Scaffold**, ajoutons un **FloatingActionButton** uniquement sur la liste des Tasks.

```
floatingActionButton: _selectedIndex==0?FloatingActionButton(
          onPressed: (){},
          child: const Icon(Icons.add),):const SizedBox.shrink(),
```

Le click sur ce bouton doit permettre de naviguer vers un nouveau widget qui pourra ajouter une Task via un simple bouton.

Le widget AddTask

Voici le code de notre widget:

```
class AddTask extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Add Task'),
      ),
      body: Center(
        child: ElevatedButton(
          style: ElevatedButton.styleFrom(
            foregroundColor: Colors.redAccent,
            backgroundColor: Colors.lightBlue,
          ),
          onPressed: () {},
          child: const Text("Add Task"),
        ),
      ),
    ) ;
  }
}
```

Navigation

La navigation dans une application Flutter n'est pas forcément évidente à gérer. Il existe de nombreux package facilitant la mise en place d'un système de routage pour des applications. Dans notre cas qui est très simple, nous allons utiliser **Navigator** avec les méthodes **push et pop**. Le click sur le FloatingActionBouton permet d'accéder au Widget **AddTask**.

```
onPressed: (){
            Navigator.push(context, MaterialPageRoute(
              builder: (context) => AddTask(),
            )
            );
          },
```

Le click sur le bouton **Add Task** ajoute une Task via le ViewModel puis renvoie à la liste des Tasks.

```
onPressed: () {
            context.read<TaskViewModel>().addTask(Task.newTask());
            Navigator.pop(context);
          }
```