Utilisation de Riverpod

Introduction :

Avant toutes choses, merci de parcourir la documentation de la librairie : https://riverpod.dev/fr/

Riverpod est une librairie de gestion d'état pour Flutter qui offre des fonctionnalités avancées pour gérer l'état d'une application de manière efficace et réactive. Dans Riverpod, les providers sont la base. Un provider est un objet qui encapsule un état et qui permet d'écouter les changements de cet état. Il en existe plusieurs type et il est possible de les combiner.

Utilisation des providers

Il existe plusieurs types de provider et nous allons voir dans les prochaines parties comment utiliser les différents types.

Lire un provider read or watch

`ref.watch` : Permet d'obtenir la valeur d'un provider et écouter les changements, de sorte que lorsque cette valeur change, cela reconstruira le widget ou le provider qui s'est abonné à la valeur.

`ref.read` : Permet d'obtenir la valeur d'un provider tout en ignorant les changements. Cela est utile lorsque nous avons besoin de la valeur d'un provider dans un événement tel que "on click".

La règle est simple donc simple dans 99% des cas on utilise `ref.watch` dans les méthodes de build des widgets et lorsque l'on combine des providers alors que `ref.read` est utile sur des actions ponctuelles (tap, initState, …)

Singleton

Utiliser un `Provider`. C'est le plus basique de tous les providers. Il crée une valeur... Et c'est à peu près tout.

```
final stringProvider = Provider<String>(
  (ref) => 'Hello world',
);
```

La valeur peut ensuite être récupérer :

```
final stringValue = ref.watch(stringProvider);
print(stringValue); // Display Hello world
```

En général, on va utiliser ce pattern n'ont pas pour stocker des valeurs primitive (String, int, bool, …) mais plutôt pour créer des singleton pour des services par exemple.

```
final loggerServiceProvider = Provider<LoggerService>(
  (ref) => const LoggerService(),
);

class LoggerService {
```

```
  const LoggerService();

  void customPrint(String text) {
    print('Log: $text');
  }
}


/// ...
ElevatedButton(
  onPressed: () {
    final loggerService = ref.read(loggerServiceProvider);
    loggerService.customPrint('Log printed by loggerService');
  },
  child: const Text('Print'),
),
```

Cela permet ensuite de faire de l'injection de dépendances de manière élégante et testable car facilement remplaçable par des mock. Voici un exemple d'un service qui aurait besoin d'un autre service :

```
final otherServiceProvider = Provider<OtherService>(
  (ref) {
    final loggerService = ref.watch(loggerServiceProvider);
    return OtherService(loggerService: loggerService);
  },
);

class OtherService {
  const OtherService({required this.loggerService});

  final LoggerService loggerService;

  void doSomething(String text) {
    print('From OtherService :');
    loggerService.customPrint(text);
  }
}
```

StateProvider

`StateProvider` est un provider qui expose un moyen de modifier son état.

Il existe principalement pour permettre la modification de variables simple par l'interface utilisateur (enum, String, bool, int, …).

> 💡 Pour des cas plus complexe, on peut utiliser un `StateProvider` avec une classe Freezed.

Exemple d'utilisation :

```
final boolProvider = StateProvider<bool>(
  (ref) => false,
);

final countProvider = StateProvider<int>(
  (ref) => 0,
);
```

Pour un `StateProvider`, la valeur retourner dans le constructeur correspond à la valeur initiale.

La lecture se fait comme pour un `Provider` classique :

```
final count = ref.watch(countProvider);
final boolValue = ref.watch(boolProvider);
```

Pour modifier le state, il faut passer par le notifier du Provider. Le choix s'offre à nous :

```
ref.read(countProvider.notifier).state = value + 1;
ref.read(countProvider.notifier).update((state) => state + 1);
ref.read(countProvider.notifier).state++;
```

Il est possible de lire un provider ou de le modifier dans un service. Pour cela, il faut injecter en dépendance l'objet `ref`:

```
final modifierServiceProvider = Provider<ModifierService>(
  (ref) {
    return ModifierService(ref: ref);
```

```
  },
);

class ModifierService {
  const ModifierService({required this.ref});

  final Ref ref;

  void incrementValue() {
    final canIncrement = ref.read(boolProvider);
    if (canIncrement) {
      ref.read(countProvider.notifier).state++;
    }
  }
}
```

Providers composés

Il existe différents cas où on va avoir besoin de composer avec plusieurs Providers

Injection de dependance

Riverpod permet efficacement de faire de l'injection de dépendance entre différents services. Les services dont dépendent une classe peuvent être récupérer via `ref.watch`

```
final service1Provider = Provider<Service1>(
  (ref) => const Service1(),
);

class Service1 {
  const Service1();
}

final service2Provider = Provider<Service2>(
  (ref) => const Service2(),
);

class Service2 {
  const Service2();
}

final service3Provider = Provider<Service3>(
  (ref) {
    return Service3(
      service1: ref.watch(service1Provider),
      service2: ref.watch(service2Provider),
    );
  },
);
```

```
class Service3 {
  const Service3({
    required this.service1,
    required this.service2,
  });

  final Service1 service1;
  final Service2 service2;
}
```

Combiner plusieurs data

Un `Provider` simple peut être utilisé pour combiner plusieurs providers. En combinaison avec des `StateProvider` cela permet au `Provider` de recalculer sa valeur chaque fois qu'un `StateProvider`est modifié.

Dans l'exemple ci-dessous, `fullNameProvider`serait recalculer a chaque fois que le last name ou le first name évolue :

```
final firstNameProvider = StateProvider<String>(
  (ref) => '',
);

final lastNameProvider = StateProvider<String>(
  (ref) => '',
);

final fullNameProvider = Provider<String>(
  (ref) {
    final firstName = ref.watch(firstNameProvider);
    final lastName = ref.watch(lastNameProvider);
    return '$firstName $lastName'.trim();
  },
);
```

Cibler la data que l'on veut observer

Pour optimiser le nombre de rebuild d'un widget, il est important d'écouter les changement d'un provider uniquement sur la data qui nous intéresse. En effet, le widget se rebuild chaque fois que la valeur du Provider est différente (qui se base sur le `==` de la classe, donc soyez vigilant à utiliser des classes qui implémentent le `==` correctement : types primitifs, classe freezed).

Par exemple si on veut rendre actif un bouton seulement quand le `fullName` a du contenu, on a pas besoin de rebuild le widget à chaque fois que le fullName évolue mais uniquement si la variable `isNotEmpty` change.
Pour cela, 2 choix s'offre à nous :

- `select` :

```
final canValidate = ref.watch(fullNameProvider.select((value) => value.isNotEmpty));
```

- Créer un nouveau `Provider`:

```
final canValidateProvider = StateProvider<bool>(
  (ref) {
    final fullName = ref.watch(fullNameProvider);
    return fullName.isNotEmpty;
  },
);

final canValidate = ref.watch(canValidateProvider);
```

Comment choisir ? On va préférer utiliser `select` quand on a besoin de cette donnée à un seul endroit alors qu'on n'hésitera pas à créer un nouveau `Provider` si cette valeur est utilisée à plusieurs endroits dans le code.

Tips de performance :
Pour limiter le nombre de rebuild de chaque widget, il est important de respecter ces règles :
- Découper un maximum les widgets pour que les `ConsumerWidget` soit le plus petit possible afin de rebuild uniquement les widgets qui dépendent des changements de state.
- Créer des `Provider` spécifiques ou utiliser les `select` pour ne cibler que la data qui a de l'influence sur la UI.

Provider en tant qu'`InheritedWidget`

Problématique

Dans une application, certaines pages sont conditionnées par l'`id` d'un objet. C'est par exemple le cas quand on veut avoir le détail d'un item. En général, l'`id` est passé dans la navigation.
Ensuite dans la page, plusieurs widgets peuvent avoir besoin de récupérer l'id. Cela peut être pour l'afficher, pour faire une requête au serveur, pour filtrer des données …

Un débutant en Flutter aura tendance à passer cet valeur en paramètre de tous les widgets de l'arbre pour que les widgets qui en ont besoin puisse lire cette valeur. C'est contraignant et lourd à écrire ce qui va inciter le débutant à ne pas trop découper son code en petits widgets car il faudra passer la valeur à chaque fois. Ce n'est donc pas la solution optimale !
En Flutter, il existe un widget peu utiliser en pratique qui permet de régler ce problème : l'`InheritedWidget`. (https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html)

On ne va pas détailler ici comment implémenter un`InheritedWidget`. Ce qu'il faut retenir c'est que cela permet d'injecter une valeur à un niveau de l'arbre des widgets et de pouvoir la récupérer n'importe où dans les widgets fils avec la méthode `of(context)`.

Implémentation avec Riverpod

On peut faire l'équivalent d'un inherited widget en utilisant uniquement Riverpod. Pour cela, on va créer un `Provider` que l'on va override avec la valeur que l'on veut stocker à un niveau de l'arbre avec un `ProviderScope`. Tous les widgets `child` du `ProviderScope` pourront récupérer cette valeur. Si on essaye de récupérer la valeur en dehors des child du ProviderScope, l'exception `UnimplementedError` sera raised.

```
final idGetter = Provider<String>(
  (ref) => throw UnimplementedError(),
);

class InheritedProviderPage extends StatelessWidget {
  const InheritedProviderPage({Key? key, required this.id}) : super(key: key);

  final String id;

  @override
  Widget build(BuildContext context) {
    return ProviderScope(
```

```
      overrides: [
        idGetter.overrideWithValue(id),
      ],
      child: const _Layout(),
    );
  }
}
```

La lecture de la valeur de `idGetter` se fait comme avec un `Provider` classique :

```
final id = ref.watch(idGetter);
```

On aime bien utiliser ce pattern pour éviter de transmettre facilement les paramètres qui sont passer à une page. Cela allège et clarifie le code.

⚠️ Soyez vigilant :
- Les `InheritedProvider` sont plus technique que le reste, ne commencer pas à mettre des `ProviderScope` dans des `onPressed` en pensant modifier la valeur … ça n'aurait aucun sens.
- Ne pas confondre les `StateProvider` avec les `InheritedProvider`. Leur fonction est totalement différente.
- Quand vous utilisez ce pattern, vérifier que l'override dans le `ProviderScope` est bien un widget parent des child qui ont besoin de récupérer la valeur. Plus particulièrement faire attention au changement de page ou avec les dialogs.

Composition et family

La composition de `Provider` avec des `InheritedProvider` est possible mais déconseillée pour plusieurs raisons :

```
final idGetter = Provider<String>(
  (ref) => throw UnimplementedError(),
);

final uuidProvider = Provider<UuidValue>(
```

```
  (ref) {
    final id = ref.watch(idGetter);
    return UuidValue.fromList(Uuid.parse(id));
  },
  dependencies: [
    idGetter,
  ],
);
```

- Il faut manuellement spécifier le `InheritedProvider` en tant que `dependencies`
- Le `uuidProvider` (pour les même raisons que le `InheritedProvider`) peut throw une exception s'il utilisé en dehors des `child` du `ProviderScope` . Il n'est donc pas safe.
- Je suis déjà tombé sur des problèmes lié à la lib lors de composition de Provider qui sont override …

Pour les raisons cité précédemment, on va préférer utiliser les modifiers `family` si on a besoin de ces valeurs dans d'autres provider.
(https://riverpod.dev/docs/concepts/modifiers/family)

Le provider devient donc :

```
final familyUuidProvider = Provider.family<UuidValue, String>(
  (ref, id) {
    return UuidValue.fromList(Uuid.parse(id));
  },
);
```

Et la lecture se fait en lui passant l'id :

```
final id = ref.watch(idGetter);
final uuid = ref.watch(familyUuidProvider(id));
```

Travaux pratiques

Récupérer la branche tp/start.
Les questions suivantes vous guideront pour re-factoriser le code des fichiers `vehicle_manager_page.dart` et `vehicle_details_page.dart` en utilisant les concepts de Riverpod expliqués précédemment. A chaque question, vous ferez attention à

l'optimisation du nombre de rebuild. Pour faire simple, pensez à découper en petit widget 😁 Quand les questions indiquent `Créer un widget`, il faut utiliser les composants visuels déjà présents, c'est principalement du refacto.

Q1 : Utiliser un StateProvider pour showAddSection

- Remplacer le bool `showAddSection` géré via un StatefullWidget par un StateProvider (showAddSectionProvider)
- Créer un widget (_AppBarButton) qui va afficher le button correspondant à la bonne action dans l'appBar
- Créer un widget (_AddSection) qui va soit afficher la section pour ajouter un véhicule soit renvoyer une SizedBox()

A cette étape `_AddSection` devrait toujours être un StatefullWidget

Q2 : Supprimer les StatefullWidgets

- Dans `_VehicleManagerPage` utiliser un `StateProvider` pour stocker la liste de véhicules (vehiclesProvider)
- Dans `_AddSection` remplacer les variables `name`, `year` et `description` par des `StateProvider` (nameProvider, yearProvider, descriptionProvider)
- Créer un widget (_VehicleList) qui se charge d'afficher la listView des véhicules basé sur le state de `vehiclesProvider`

A ce stade, vous ne devriez plus avoir de `StatefullWidget`

> 💡 Attention quand vous utilisez Riverpod avec des listes. Le rebuild des widgets se base sur le `==` des classes. Si vous faites des add et des remove sur une liste Riverpod considère que c'est la même liste et donc ne va pas rebuild la UI.

Q3 : Créer des Services et des ViewModels

Ici on souhaite créer des classes qui seront testables facilement et qui regroupe la partie métier de l'application.

- Créer un service (DataValidatorService) qui permet de vérifier la validité des champs `name` et `year`. Puis utiliser le pattern Singleton de Riverpod pour rendre facilement disponible ce service.

```
class DataValidatorService {
  const DataValidatorService();

  bool yearIsValid(String year) {
    ...
  }

  bool nameIsValid(String name) {
    ...
  }
}
```

- Créer un ViewModel (VehicleManagerViewModel) qui permet d'ajouter ou de supprimer un véhicule. Puis utiliser le pattern Singleton de Riverpod pour rendre facilement disponible ce ViewModel.

```
class VehicleManagerViewModel {
  const VehicleManagerViewModel({
    required this.ref,
    required this.dataValidatorService,
  });

  final Ref ref;
  final DataValidatorService dataValidatorService;

  void addVehicle({
    required String name,
    required String year,
    required String description,
  }) {
    // TODO : Use dataValidatorService to validate parameters
    // and add the vehicle to the vehiclesProvider
  }

  void deleteVehicle(String id) {
    // TODO : remove the vehicle from indicated by the id
    // from the vehiclesProvider
  }
}
```

- Utiliser le `VehicleManagerViewModel` pour ajouter ou supprimer un véhicule dans les boutons correspondant.
- Créer un Provider composé (canAddProvider) qui à partir de `dataValidatorService`, `nameProvider`, `yearProvider` retourne un bool qui indique si on peut ajouter un véhicule.
- Créer un widget (_AddButton) qui utilise le `canAddProvider` pour désactiver le bouton d'ajout.

Q4 : Utiliser un inherited provider

- Sur la page de détails (VehicleDetailsPage), utiliser le pattern `InheritedProvider` pour transmettre le `vehicleId` dans toute la page (vehicleIdGetter).

Q5 : Utiliser le modifier family

- Créer un `Provider.family` (vehicleProvider) qui à partir d'une String `vehicleId` et du `vehiclesProvider` permet de renvoyer le `Vehicle` associé.
- Supprimer la variable globale `currentVehicles`
- Utiliser `vehicleProvider` en combinaison avec des `select` dans les widgets qui affichent le détail d'un véhicule (_NameText, _YearText, _DescriptionText)