Flutterで
いい感じに状態管理

RIVERPODを採用するに至った理由

自己紹介

- 某クラウドベンダーでプロトタイピング　エンゲージメントに従事
  - A社のクラウドを使う時にブロッカーをクリアして導入までの時間短縮
- Comictribes.comソフトウェア開発グループを主宰
  - メインプロダクトはコミケカタログブラウザ
- Android、iOSのモバイルには以前から関心あり
- 最近すごく興味を持っているのはMSX0

Agenda

アプリのテクノロジー負債
思わず目移りする技術たち
自分の中のニーズを見つめ直す

状態保持で何を保持するか？
何を採用することにしたか？
フレームワークの比較
フレームワークの選定

Riverpodでできること
具体的な使い方
CQRSとイベントソーシング
ListViewでCQRSを実装してみる
デバッグに便利なテクニック
最後に

アプリのテクノロジー負債(重いよね・・・)

Comictribesの場合：開発開始は**2010年**

そもそも言語が古い

iOSはobjective-c、AndroidはJava

構造がそもそもレガシー

Fragmentベースのコード

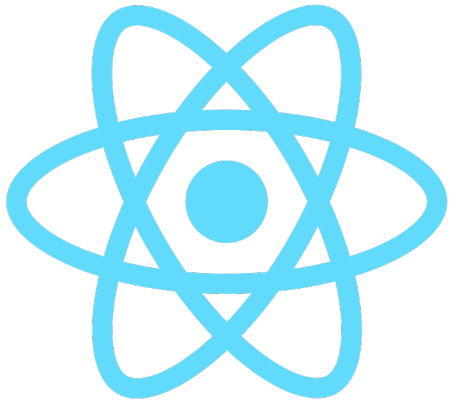
画面更新系はAsyncTaskに依存

美味しそうなスパゲッティ

コードと画面更新系の全てをFat Fragmentで記述している

思わず目移りする技術たち

- **基本アーキテクチャ**
  - MVC,MVVM,DDD,Clean-Architecture etc。。。
- **フレームワーク**
  - Flutter、React Native、.Net系技術(Xamarin、MAUI)
- **状態管理(Flutterだけでも結構ある）**
- **非同期フレームワーク**
- **Stream中心でデザインするか**



どれを採用したらいいか迷ってしまう・・・・

自分の中のニーズを見つめ直す

技術に目移りするよりも、何が重荷になっていたかを確認、優先度をつけてみる

- **iOSとAndroidの両対応**
  - 実質別アプリをメンテしてる感じ
- **アプリケーションデータの状態保持とライフサイクル管理**
  - 最新かつメンテされてる状態管理を使いたい
  - UIのニーズに応えられる状態管理を使いたい
- **ロジックと画面更新の疎結合**
  - Tab画面を跨いだフローティングウィンドウを持つアーキテクチャを作れる
  - 非同期のDB処理、API処理のエラー発生時の更新挙動をシンプルに実装したい
  - 画面更新は最低限にしたい。ListViewとかGridViewとか・・・
  - データ更新での画面更新を簡潔に記述したい

状態保持で何を保持するか？

- アプリのデータの保持に使う
  - グローバルな共通API、レポジトリのエントリーポイント
    - DBアクセスやSharedPreferenceなどのレポジトリのハンドラを保持
    - インターネット上のRest APIを利用する際のLimitコントロール
    - 更新を通知して表示データを更新するトリガー
  - グローバルな共通データ保持
    - 起動時の生成データの保持
    - 画面を跨いだ情報の保持、更新の通知
  - ウィジェットの表示データ保持
    - できるだけウィジェットツリーの枝でデータを保持して更新を減らしたい
    - ListViewなどで、個々のアイテムの情報保持と更新対象のみの部分更新

何を採用することにしたか？

- **Flutterの採用**
  - Material DesignベースでUXを統一したいと考えていたのでこれは早い段階で決定
- **基本アーキテクチャはあえて決めない**
  - やりたいことは関心の分離、DIしやすいものを選ぶ
  - 作りやすいことを重視
  - 状態保持フレームワークの設計思想重視
- **状態管理**
  - 継続性を重視し、できるだけ人気のフレームワークを使用
  - 平易な書き方ができるか
    - Riverpodを採用
- **ロジックと画面更新の疎結合**
  - 画面更新は状態変更した時に発生するので、状態保持に依存
  - CQRSを実現するイベントソーシングを実装
    - Riverpod (Notifier Providerを使用)

フレームワークの比較（人気重視）

	更新日時	Star	Like	Flutter Fovorite	URL
Provider	2023/11/10	4900	9121	○	https://github.com/rrousselGit/provider
Riverpod	2023/10/31	5200	2800	○	https://github.com/rrousselGit/riverpod
bloc	2023/11/7	1100	6123	○	https://github.com/felangel/bloc
RxDart	2023/10/26	3300	2328	○	https://github.com/ReactiveX/rxdart
mobx	2023/10/31	2300	1100	○	https://github.com/mobxjs/mobx.dart
getX	2023/11/3	9300	13168	×	https://github.com/jonataslaw/getx
GetIt	2023/9/25	1200	3497	×	https://github.com/fluttercommunity/get_it

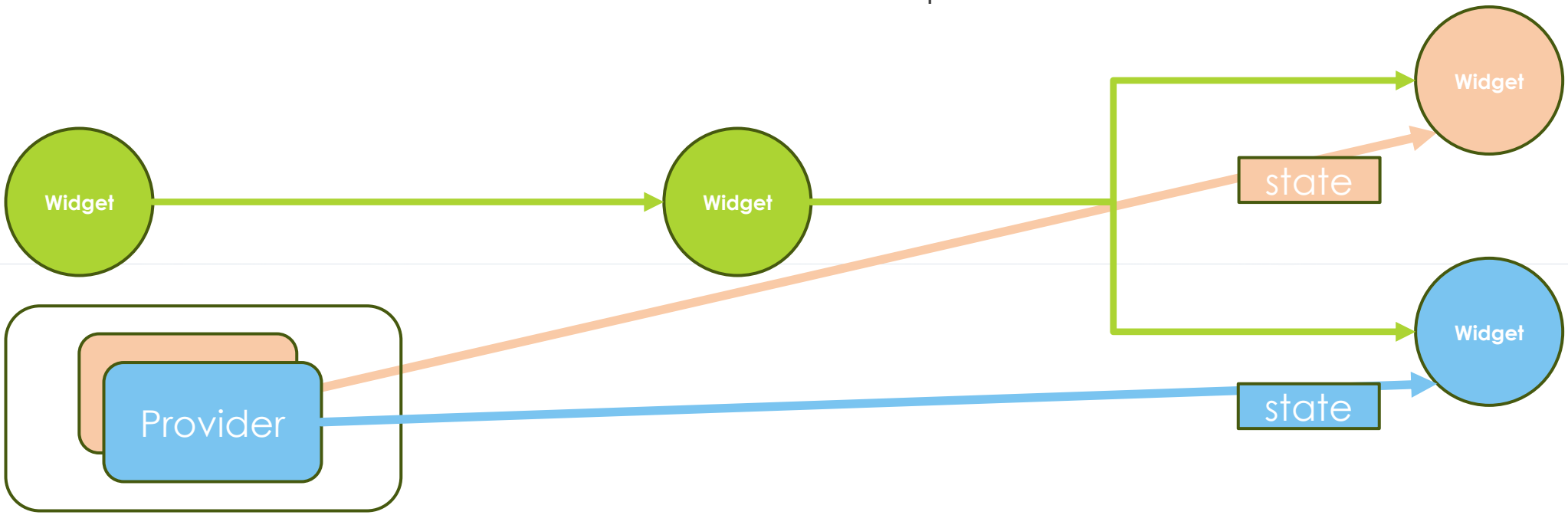
フレームワークの選定（平易さ）

	概要
Provider	InheritedWidgetをより使いやすく、より再利用しやすくするためのラッパー
Riverpod	Providerと同一作者。ビジネスロジックの記述と、非同期処理に使用できるリアクティブ キャッシュ フレームワーク。状態保持にも利用可能
bloc	Blocアーキテクチャの考え方に基づいてStreamベースのデータ処理とCubitによる状態更新のemitをサポートするフレームワーク。Providerに依存
RxDart	いわゆるReactiveXのDartでの実装。Streamを拡張してさまざまなストリーム処理ができる。Blocライブラリのような処理を自前で作り込む場合に使う。
mobx	MobXは、アプリケーションのリアクティブ・データとUI（または任意のオブザーバー）を簡単に接続できるステート管理ライブラリ。アノテーションを積極的に利用。Providerに依存

できるだけ平易な書き方を志向したい（Stream処理は複雑な書き方ができてしまう）
アノテーションが多いのは、学習コストが多く望ましくない
＝＞Riverpodを選定

Riverpodでできること

- 画面表示情報などのステート情報を保存、取得、更新できる
- ステートを更新するとUIがリビルドされる
- ウィジェットの一部だけ更新することもできる
- ListItemなどで便利な引数付きのProviderやautoDisposeが利用できる



具体的な使い方(non annotation)


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

import
'package:riverpod_annotation/riverpod_annotation.dart';

part 'main.g.dart';


final helloWorldProvider = Provider((ref) => 'Hello World');

void main(){
  runApp(
    ProviderScope(
      child: MyApp(),
    ));}
```

定義：ProviderをGlobal scopeで関数定義

スコープ有効化：Providerの検索スコープ設定

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer(
        builder: (context, ref, _) {
          final helloWorld = ref.watch(helloWorldProvider);
          return MaterialApp(
              home: Scaffold(
                body: Center(
                  child: Text(helloWorld),
                ),
              ));
        });
  }}
```

監視設定：helloWorldを監視

helloWorldを監視して変更したらビルド

Riverpodの使い方（ non annotation ）

Widgetと一緒に使う場合

- 定義方法
  - グローバルで定義されたprovider変数で定義
  - .familyで引数付きのproviderや、.autoDisposeでWidgetの削除に同期して削除する挙動を選べる
- 利用可能な場所
  - ConsumerWidgetやWidgetツリーの中のConsumerから利用
- 利用方法
  - RiverpodのウィジェットにBuildメソッドの引数に含まれるWidgetRefとproviderインスタンスから状態取得。状態の取得には、呼び出した時点の状態を取得するProvider.readと動的な状態変化に追従するwatchの２種類がある。ほとんどのケースでwatchで問題ない。
  - Widgetで状態更新をobserveしたい場合はステートをwatchで状態取得
    XxxState state=ref.watch(xxxProvider);
  - StateNotifier等を継承して状態を保持している場合、クラス内のステート更新関数にアクセスしたい場合がある。StateNotifier stateNotifier=ref.watch(xxxProvider.Notifier);でインスタンスを取得できる

具体的な使い方（最近の書き方）


```dart
import 'package:flutter/material.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';
import
'package:riverpod_annotation/riverpod_annotation.dart'
;
import 'package:riverpod/riverpod.dart';

@riverpod
final String helloWorld = ((ref) => 'Hello World');
```

定義：ProviderをGlobal scopeで関数定義

```dart
void main() {
  runApp(
      ProviderScope(
        child: MyApp(),
      ));}
```

スコープ有効化：Providerの検索スコープ設定

```dart
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
 return Consumer(
     builder: (context, ref, _) {
      final helloWorld = ref.watch(helloWorldProvider);
      return MaterialApp(
         home: Scaffold(
           body: Center(
             child: Text(helloWorld),
           ),
         ));
     });
}}
```

監視設定：helloWorldを監視

helloWorldを監視して変更したらビルドが走る

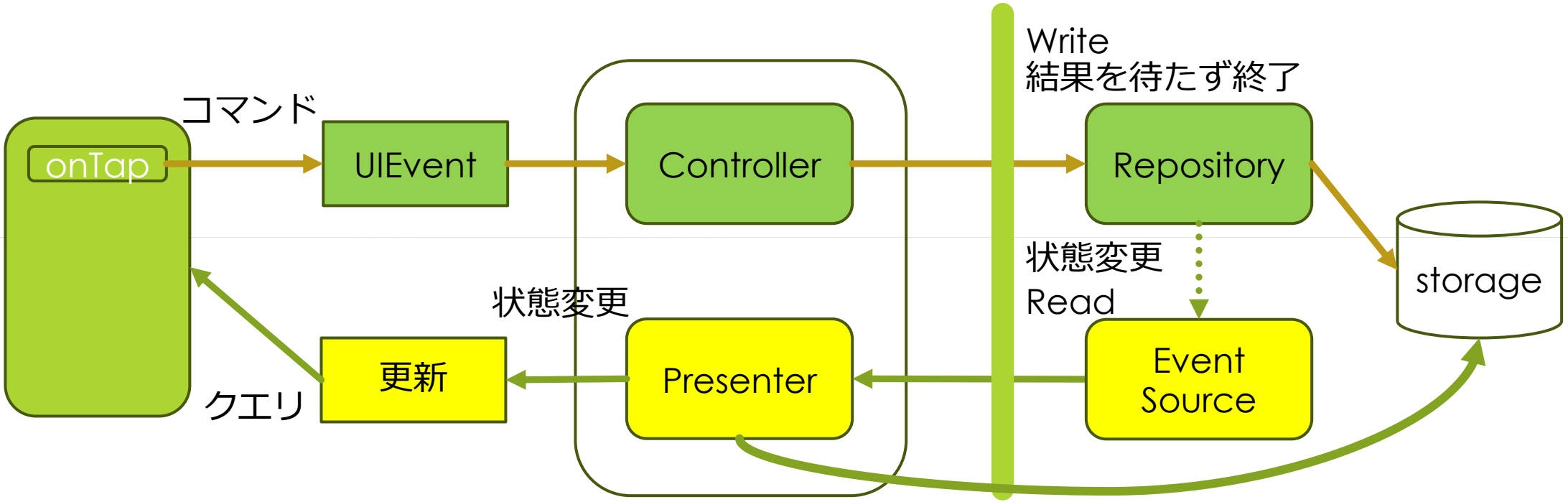
Riverpodの使い方(2.0以降)

Widgetと一緒に使う場合

- 定義方法
  - グローバルでprovider変数を生成して定義
  - .familyで引数付きのproviderや、.autoDisposeでWidgetの削除に同期して削除する挙動を選べる
  - @riverpodアノテーションで{クラス名}providerが自動生成され、buildの引数で.familyを自動判定
  - Notifierの.familyはアノテーションで作る前提となる
- 利用可能な場所
  - ConsumerWidgetやWidgetツリーの中のConsumerから利用
- 利用方法
  - RiverpodのウィジェットにBuildメソッドの引数に含まれるWidgetRefとproviderインスタンスから状態取得。状態の取得には、呼び出した時点の状態を取得するProvider.readと動的な状態を取得するwatchの２種類がある。ほとんどのケースでwatchで問題ない。
  - Widgetで状態更新をobserveしたい場合はステートをwatchで状態取得
    XxxState state=ref.watch(xxxProvider);
  - StateNotifier等を継承して状態を保持している場合、クラス内のステート更新関数にアクセスしたい場合がある。StateNotifier stateNotifier=ref.watch(xxxProvider.Notifier);でインスタンスを取得できる

CQRSとイベントソーシング

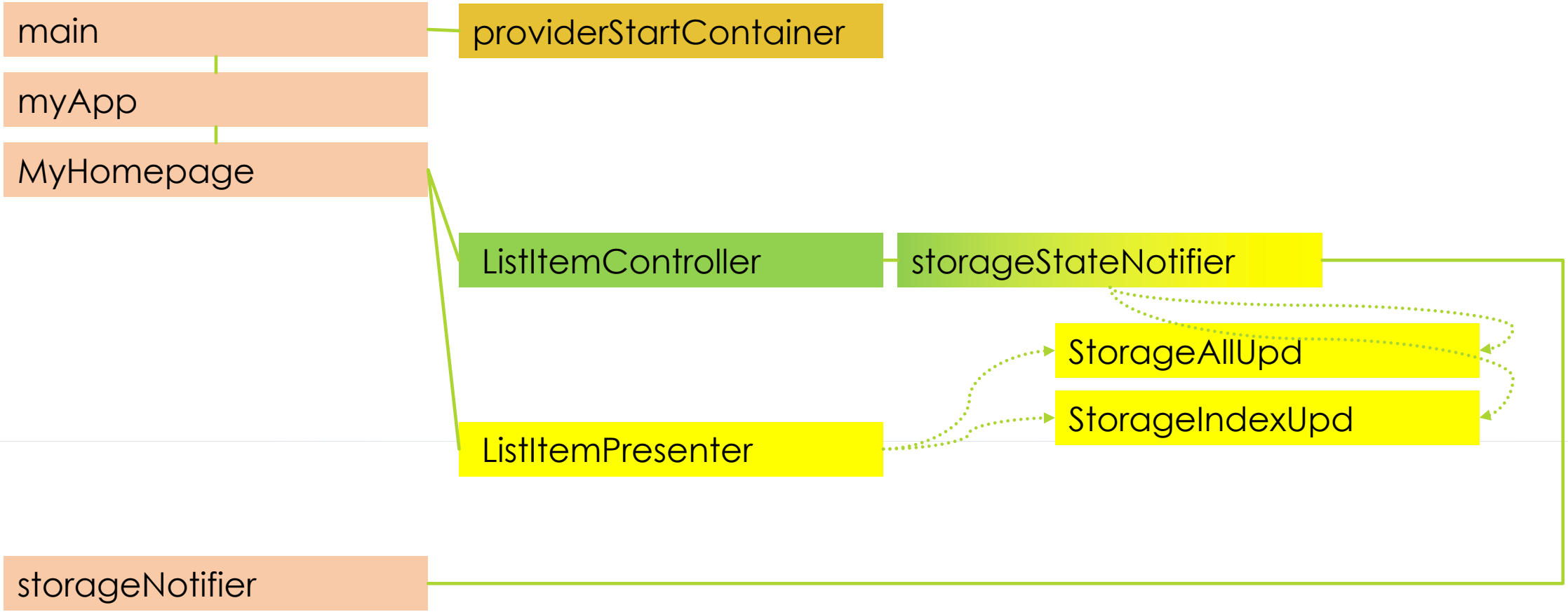
- CQRS（コマンドクエリ責務分離）
  - Commands システムの状態を変更するが、値を返す必要がない
  - Queries　結果を返すが、システムの状態の変更は行わない（副作用を伴わない）
- イベントソーシング
  - あるドメインで発生したイベントを他のドメインに供給するデザインパターン



CQRSとイベントソーシングの実装

- CQRSの実装
  - Riverpod でUIの動作を受け取るControllerをメソッドとして実装
  - UIイベントを受けてレポジトリへ更新を行うが、書き込み結果を待たないで終了
- イベントソーシングの実施
  - RiverpodでRepositoryの状態変化を伝達するイベントソースを作成
  - Riverpodでイベントソースを監視するPresenterを実装。更新が発生したらDBを読んで状態変更
- 画面更新の連携
  - UIはPresenterを監視して状態変更が発生したら画面更新

ListViewでCQRSを実装してみる



CQRSデモ

全体の動作

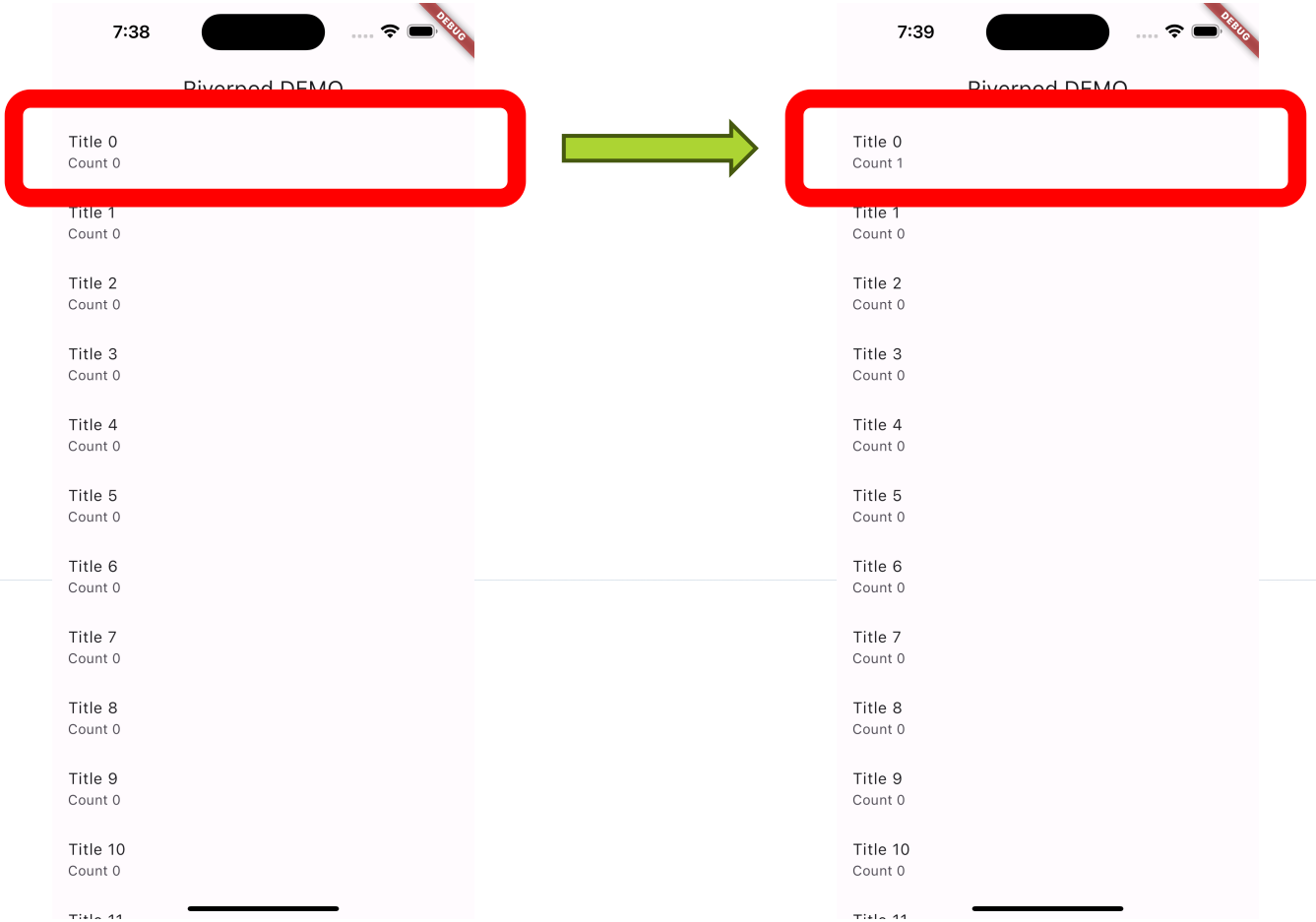
- 書き込みボタン->レポジトリに書き込み依頼->UI更新せず終了
- レポジトリ書き込み、書き込み終了でイベント発生
- レポジトリのイベント購読->自分に関連したイベントなら再読み込み、画面更新

結果

- 書き込みと読み込みが分離されていて、レポジトリの更新をトリガーに画面更新される

デモ画面

リストアイテムをクリックすると、そのアイテムのみカウンターが増えて更新される



ListViewでCQRSを実装してみる


```dart
import 'package:flutter/material.dart';
import 'package:freezed_annotation/freezed_annotation.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';
import 'package:riverpod_annotation/riverpod_annotation.dart';
part 'main.freezed.dart';
part 'main.g.dart';

void main()  {
  final widgetsBinding = WidgetsFlutterBinding.ensureInitialized();
  final providerStartContainer= ProviderContainer();
  final storageStateNotifier=providerStartContainer.read(storageStateProvider.notifier);
  storageStateNotifier.clear();
runApp(
    ProviderScope(  //プロバイダー監視開始
        parent: providerStartContainer,
        child:const MyApp()
      ));}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
   //APPを定義
        return MaterialApp(
      title: 'RiverPod Demo',
      theme: ThemeData(useMaterial3: true),
      home: const MyHomePage(),
    );}}

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key});
  @override
  Widget build(BuildContext context) {
   return Scaffold(
      appBar: AppBar(
        title: Text('Riverpod DEMO'),
      ),
      body: Center(
        child: Consumer(builder: (BuildContext context, WidgetRef ref, Widget? _){
          final storageAllUpdState=ref.watch(storageAllUpdProvider);
          final storageStateController=ref.watch(storageStateProvider.notifier);
          return ListView.builder(
              itemCount: storageStateController.getCount(),
              itemBuilder: (BuildContext context, int index) {
                return
                    Consumer(builder: (BuildContext context, WidgetRef ref, Widget? _) {
                      Final listItemStateController =
                          ref.watch(listItemControllerProvider(index).notifier);
                      final listItemState = ref.watch(listItemPresenterProvider(index));

                debugPrint("$index screen rebuild");
                return ListTile(
                  title: Text("Title ${listItemState.title}"),
                  subtitle: Text("Count ${listItemState.subTitle}"),
                  onTap: () {
                    listItemStateController.onTap(index);
                  },
                );});});},));}
}
```

ListViewでCQRSを実装してみる


```
@freezed
class ListItem with _$ListItem {
  const factory ListItem({
    required String title,
    required String subTitle,
  }) =_ListItem;
}
@riverpod
class ListItemController extends _$ListItemController{
  @override
  void build(int index) {}
  void onTap(int index) {
    final storageStateNotifier=ref.watch(storageStateProvider.notifier);
    var dbItem=storageStateNotifier.read(index);
    dbItem=dbItem.copyWith(value: dbItem.value+1);
    storageStateNotifier.update(index, dbItem);
  }
}
@riverpod
class ListItemPresenter extends _$ListItemPresenter{
  @override
  ListItem build(int index) {
    final  storageAllUpdState=ref.watch(storageAllUpdProvider);
    final  storageIndexUpdState=ref.watch(storageIndexUpdProvider(index));
    final StorageStateNotifier storageStateNotifier=ref.watch(storageStateProvider.notifier);
    final item=storageStateNotifier.read(index);
    return ListItem(title: item.title, subTitle: item.value.toString());//freezedなので必ず更新
  }
}
@riverpod
class StorageAllUpd extends _$StorageAllUpd {
  @override
  StorageStateValue build(){return StorageStateValue(state: -1);}
  void changed() {state=StorageStateValue(state: -1);}
}

@riverpod
class StorageIndexUpd extends _$StorageIndexUpd {
  @override
  StorageStateValue build(int index){return StorageStateValue(state: index);}
  void changed(int index) {state=StorageStateValue(state: index);}
}
```

ListViewでCQRSを実装してみる


```
final storageStateProvider = AutoDisposeNotifierProvider<StorageStateNotifier,StorageStateValue>(
    StorageStateNotifier.new
);
final storageProvider = NotifierProvider<StorageNotifier,bool>(
   StorageNotifier.new
);

@freezed
class StorageStateValue with _$StorageStateValue {
 const factory StorageStateValue({required int state,}) =_StorageStateValue;
}



class StorageStateNotifier extends AutoDisposeNotifier<StorageStateValue>{
 @override
 StorageStateValue build() {return StorageStateValue(state:-1);}

 void clear() {
  var storageState=ref.watch(storageProvider.notifier).clear();
  ref.watch(storageAllUpdProvider.notifier).changed();
 }
 void create (DbItem dbItem){
  ref.watch(storageProvider.notifier).create(dbItem);
  ref.watch(storageAllUpdProvider.notifier).changed();
 }
 DbItem read (int index) {
  return ref.watch(storageProvider.notifier).read(index);
 }
 void update(int index,DbItem dbItem) {
  ref.watch(storageProvider.notifier).update(index,dbItem);
  ref.watch(storageIndexUpdProvider(index).notifier).changed(index);
 }
 void delete (int index) {
  ref.watch(storageProvider.notifier).delete(index);
  ref.watch(storageAllUpdProvider.notifier).changed();
 }
 int getCount() {return ref.watch(storageProvider.notifier).getCount();}
}
@freezed
class DbItem with _$DbItem {
 const factory DbItem({
  required String title,
  required int value,
 }) =_DbItem;
}
//アイテムリストのストレージ
//通常不揮発ストレージに入れるが、今回はオンメモリでシミュレート
class StorageNotifier extends Notifier<bool>{
 final List<DbItem> _dbItems=[];
 @override
 bool build() {
  clear();
  return true;
 }
 void clear() {
  _dbItems.clear();
  for (int i=0;i<20;i++){
   _dbItems.add(DbItem(title:i.toString(),value:0));
  }}
 void create (DbItem dbItem){_dbItems.add(dbItem);}
 DbItem read (int index) {return _dbItems[index];}
 void update(int index,DbItem dbItem) {_dbItems[index]=dbItem;}
 void delete (int index) {_dbItems.removeAt(index);}
 int getCount() {return _dbItems.length;}
}
```

デバッグに便利なテクニック

Observersを定義すると、Riverpodのライフサイクルイベントが表示できる

```
void main() {
  runApp(
     ProviderScope(
        child: MyApp(),
        observers: [MyObservers]
     ));}


class MyObserver extends ProviderObserver {
  @override
  void didAddProvider(
     ProviderBase<Object?> provider,
     Object? value,
     ProviderContainer container,) {
   debugPrint('Provider $provider was initialized with $value');
  }
  @override
  void didDisposeProvider(
     ProviderBase<Object?> provider,
     ProviderContainer container,) {
   debugPrint('Provider $provider was disposed');
  }
  @override
  void didUpdateProvider(
     ProviderBase<Object?> provider,
     Object? previousValue,
     Object? newValue,
     ProviderContainer container,) {
   debugPrint('Provider $provider updated from $previousValue
to $newValue');
  }
  @override  void providerDidFail(
     ProviderBase<Object?> provider,
     Object error,
     StackTrace stackTrace,
     ProviderContainer container,) {
   debugPrint('Provider $provider threw $error at $stackTrace');
  }
}
```

最後に

Riverpodを使うといい感じに状態管理ができる

状態管理（＋CQRS＋UI更新）で必要十分な画面更新を実装可能

デバッグの可視化は重要