Cubit

→ Bloc'un daha basit hali

Model → Veriyi temsil eden sınıf

Repository → Veriyi dış kaynaklardan (API, veritabanı vb.) alma ve işleme işlemlerinin yapıldığı katman

Cubit → Durumu yöneten sınıf

Ui → (View) → Cubitten gelen durumu dinleyerek kullanıcı arayüzünü yöneten katman

Model →

```
class User {
  final int id;
  final String name;
  User ({required this.id, required this.name});
  //API'den alınan veriyi user modeline dönüştürmek için factory metodu
  factory User.fromJson (Map<String, dynamic> json) {
    return User(
      id: json['id'],
      name: json['name'],
    );
  }
}
```

Jsondan User nesnesi oluşturucak factory metodu

API'den gelen Json verisini user modeline dönüştürür

→ Jsondaki 'id' alanını int olarak al

→ Jsondaki 'name' alanını String olarak al

Repository →

```
class UserRepository {
  //Burada API isteği simüle edilmiştir.
  Future<List<User>> fetchUsers() async {
    await Future.delayed(Duration(seconds: 2)); //API isteği için simülasyon
    return [
      User(id: 1, name: 'John Doe'),
      User(id: 2, name: 'Jane Smith'),
    ];
  }
}
```

Kullanıcı listesini temsil eden bir Future döndürür

Kullanıcı listesini asenkron olarak getiren metod

→ Sabit bir kullanıcı listesi döndürür

Gerçek uygulamalarda bu veri genellikle API'den alınır

Cubit →

```
import 'package:futter_bloc/flutter_bloc.dart';
class UserCubit extends Cubit<List<User>> {
  final UserRepository userRepository;
  UserCubit(this.userRepository) : super([]);
  Future<void> fetchUsers() async {
    try {
      emit([]);
      final users = await UserRepository.fetchUsers();
      emit(users);
    } catch(e) {
      emit([]);
    }
  }
```

UserCubit sınıfı, Cubiti extend eder ve state olarak List<User> kullanır

→ UserRepository dependency'si veri kaynağına erişim sağlar

başlangıç durumu boş bir kullanıcı listesi

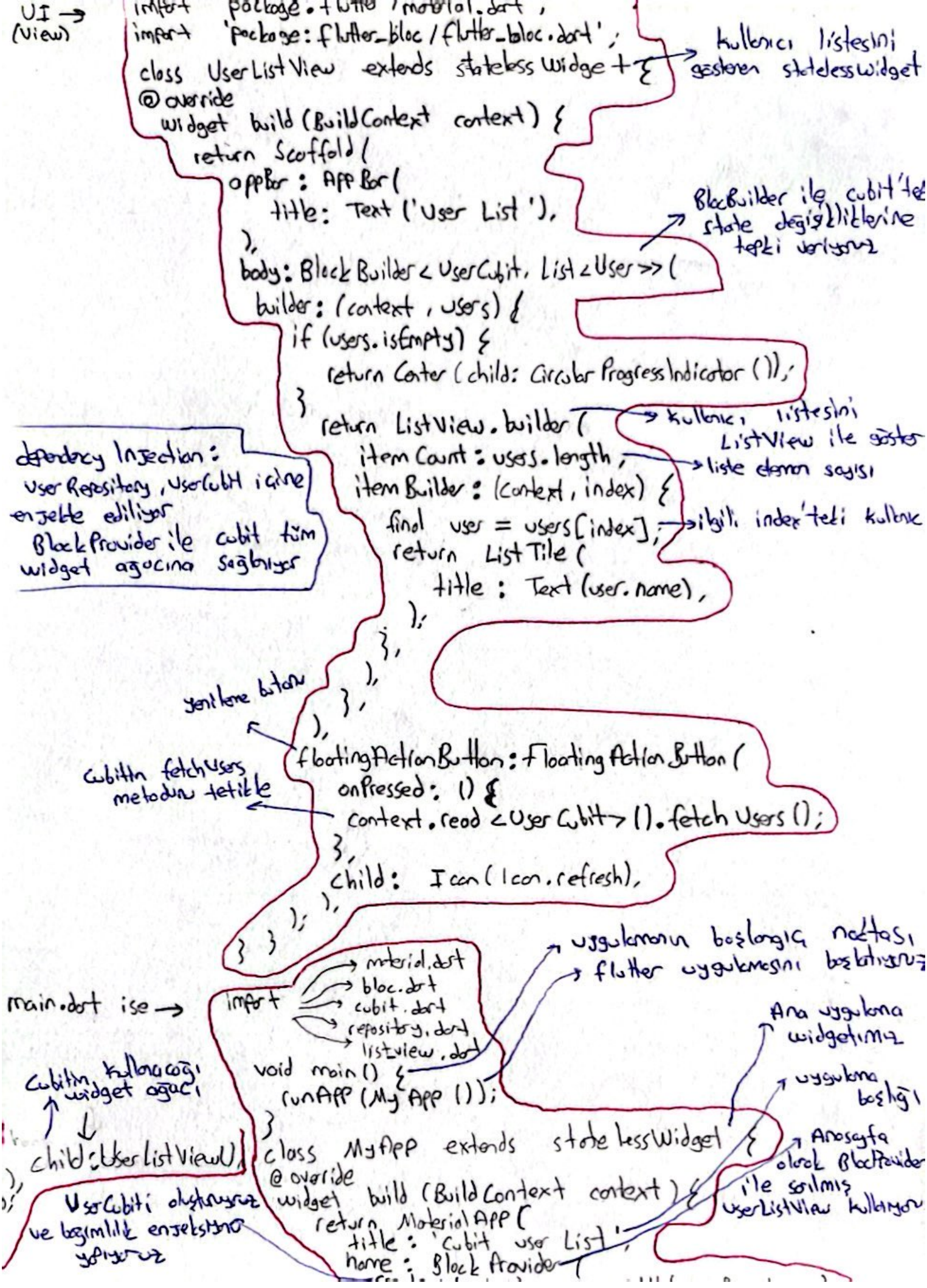
kullanıcıları fetch eden ve durumu güncelleyen method

UserRepository dependency injection ile alınır

→ boş liste ile durumu başlatıyoruz (veri yüklenirken boş ekran gösterilir)

→ kullanıcılar başarıyla alınırsa durumu güncelle

→ hata durumunda boş listeyi emit edebiliriz (hata durumu için farklı bir state kullanılabilir)


UI → (View)
import 'package:flutter_bloc/flutter_bloc.dart';
class UserListView extends StatelessWidget {
kullanıcı listesini gösteren statelessWidget
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('User List'),
),
body: BlocBuilder<UserCubit, List<User>>(
BlocBuilder ile cubit'teki state değişikliklerine tepki veriyoruz
builder: (context, users) {
if (users.isEmpty) {
return Center(child: CircularProgressIndicator());
}
return ListView.builder(
kullanıcı listesini ListView ile göster
itemCount: users.length,
liste eleman sayısı
itemBuilder: (context, index) {
final user = users[index];
ilgili index'teki kullanıcı
return ListTile(
title: Text(user.name),
);
},
);
},
);
dependency Injection: UserRepository, UserCubit içine enjekte ediliyor
BlocProvider ile cubit tüm widget ağacına sağlanıyor
yenileme butonu
floatingActionButton: FloatingActionButton(
Cubit'in fetchUsers metodunu tetikle
onPressed: () {
context.read<UserCubit>().fetchUsers();
},
child: Icon(Icons.refresh),
);
}
}
main.dart ise →
import
material.dart
bloc.dart
cubit.dart
repository.dart
listview.dart
uygulamanın başlangıç noktası
flutter uygulamasını başlatıyoruz
void main() {
runApp(MyApp());
}
Ana uygulama widgetımız
uygulama başlığı
Cubit'in kullanacağı widget ağacı
child: UserListView(),
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Cubit User List',
home: BlocProvider(
Anasayfa olarak BlocProvider ile sarılmış UserListView kullanıyor
UserCubit'i oluşturuyoruz ve bağımlılık enjekte ediyoruz