Flutter Durum Yönetimi Raporu

Flutter uygulamalarında durum yönetimi, kullanıcı arayüzünün güncel ve tutarlı kalmasını sağlayan en önemli kavramlardan biridir. Bu raporda, Flutter'da kullanılan çeşitli durum yönetimi yaklaşımlarını ve pratikte nasıl uygulandıklarını inceleyeceğiz.

İçindekiler

1. Durum Yönetimi Nedir?

Durum (state), bir uygulamanın belirli bir zamandaki veri durumunu ifade eder. Durum yönetimi ise bu verilerin nasıl saklanacağını, güncelleştirileceğini ve UI elementlerine nasıl yansıtılacağını belirleyen süreçtir.

Flutter'da durum değişiklikleri, kullanıcı arayüzünün yeniden oluşturulmasını tetikler. Doğru durum yönetimi, performanslı ve tutarlı bir kullanıcı deneyimi sağlar.

2. Flutter'da Temel Durum Yönetimi

StatelessWidget ve StatefulWidget

Flutter'da widget'lar iki ana kategoriye ayrılır:

- **StatelessWidget**: Değişmeyen durumlar için kullanılır. Bir kez oluşturulduktan sonra değişmez.
- **StatefulWidget**: Yaşam döngüsü boyunca değişebilen durumlar için kullanılır.

```
// Basit bir sayaç uygulaması için StatefulWidget örneği
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Sayaç Örneği',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Sayaç Uygulaması'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key:
key);
  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'Butona basma sayısı:',
            ),
```

```
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Artır',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

setState()

StatefulWidget'ların en temel durum yönetimi mekanizması `setState()` fonksiyonudur. Bu metod, widget'ın durumunun değiştiğini Flutter'a bildirir ve UI'ın yeniden oluşturulmasını sağlar.

3. Popüler Durum Yönetimi Çözümleri

Provider

Provider, InheritedWidget'ı daha kullanışlı hale getiren bir kütüphanedir. Durum yönetimini kolaylaştırır ve widget ağacının alt kısımlarına veri iletmeyi sağlar.

```
// Provider örneği
class CounterModel extends ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}

// Kullanım
void main() {
  runApp(
    ChangeNotifierProvider(
```

```
      create: (context) => CounterModel(),
      child: MyApp(),
    ),
  );
}

// Widget içinde kullanım
Consumer<CounterModel>(
  builder: (context, counter, child) => Text('${counter.count}'),
)
```

Bloc / Cubit

Bloc (Business Logic Component), durum yönetimini stream'ler kullanarak gerçekleştiren bir yaklaşımdır. Cubit ise Bloc'un daha basitleştirilmiş versiyonudur.

```
// Cubit örneği
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
  void decrement() => emit(state - 1);
}

// Kullanım
BlocProvider(
  create: (context) => CounterCubit(),
  child: CounterPage(),
)

// Widget içinde kullanım
BlocBuilder<CounterCubit, int>(
  builder: (context, count) {
    return Text('$count');
  },
)
```

GetX

GetX, durum yönetimi, dependency injection ve route yönetimini birleştiren hafif ve güçlü bir çözümdür.

```
// GetX Controller örneği
class CounterController extends GetxController {
  var count = 0.obs;
```

```
  void increment() => count++;
}

// Kullanım
final CounterController c = Get.put(CounterController());

// Widget içinde reaktif kullanım
Obx(() => Text('${c.count}'))
```

Riverpod

Provider'ın geliştirilmiş bir versiyonu olan Riverpod, bağımlılık enjeksiyonu ve durum yönetimini sağlar.

```
// Riverpod örneği
final counterProvider = StateNotifierProvider<CounterNotifier,
int>((ref) {
  return CounterNotifier();
});

class CounterNotifier extends StateNotifier<int> {
  CounterNotifier() : super(0);
  void increment() => state++;
}

// Kullanım
Consumer(
  builder: (context, ref, child) {
    final count = ref.watch(counterProvider);
    return Text('$count');
  },
)
```

4. Hangi Durum Yönetimi Yaklaşımını Seçmeli?

Durum yönetimi yaklaşımı seçimi, projenin karmaşıklığına ve ihtiyaçlarına bağlıdır:

- **Küçük uygulamalar için**: setState() veya Provider
- **Orta büyüklükte uygulamalar için**: Provider, GetX veya Riverpod
- **Büyük ve karmaşık uygulamalar için**: Bloc, Riverpod veya Redux

5. Örneklerle Durum Yönetimi

Alışveriş Sepeti Örneği (Provider)

```
// shopping_cart_model.dart
import 'package:flutter/foundation.dart';

class Product {
  final String id;
  final String name;
  final double price;

  Product({required this.id, required this.name, required
this.price});
}

class CartModel extends ChangeNotifier {
  final List<Product> _items = [];

  List<Product> get items => _items;

  double get totalPrice => _items.fold(0, (sum, item) => sum +
item.price);

  void add(Product product) {
    _items.add(product);
    notifyListeners();
  }

  void remove(Product product) {
    _items.remove(product);
    notifyListeners();
  }
}

// main.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'shopping_cart_model.dart';

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
```

```
      child: const MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Alışveriş Sepeti Örneği',
      home: ProductListScreen(),
    );
  }
}

class ProductListScreen extends StatelessWidget {
  final List<Product> availableProducts = [
    Product(id: '1', name: 'Laptop', price: 12000),
    Product(id: '2', name: 'Telefon', price: 8000),
    Product(id: '3', name: 'Tablet', price: 5000),
  ];

  ProductListScreen({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Ürünler'),
        actions: [
          IconButton(
            icon: const Icon(Icons.shopping_cart),
            onPressed: () {
              Navigator.push(
                context,
                MaterialPageRoute(builder: (context) =>
CartScreen()),
              );
            },
          ),
        ],
      ),
```

```
      body: ListView.builder(
        itemCount: availableProducts.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(availableProducts[index].name),
            subtitle: Text('${availableProducts[index].price}
TL'),
            trailing: IconButton(
              icon: const Icon(Icons.add_shopping_cart),
              onPressed: () {
                Provider.of<CartModel>(context, listen: false)
                    .add(availableProducts[index]);
                ScaffoldMessenger.of(context).showSnackBar(
                  SnackBar(
                    content: Text('$
{availableProducts[index].name} sepete eklendi'),
                    duration: const Duration(seconds: 1),
                  ),
                );
              },
            ),
          );
        },
      ),
    );
  }
}

class CartScreen extends StatelessWidget {
  const CartScreen({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Sepetim'),
      ),
      body: Consumer<CartModel>(
        builder: (context, cart, child) {
          return Column(
            children: [
              Expanded(
                child: cart.items.isEmpty
                    ? const Center(child: Text('Sepetiniz boş'))
```

```
                    : ListView.builder(
                        itemCount: cart.items.length,
                        itemBuilder: (context, index) {
                          return ListTile(
                            title: Text(cart.items[index].name),
                            subtitle: Text('$
{cart.items[index].price} TL'),
                            trailing: IconButton(
                              icon: const
Icon(Icons.remove_shopping_cart),
                              onPressed: () {
                                cart.remove(cart.items[index]);
                              },
                            ),
                          );
                        },
                      ),
              ),
              Container(
                padding: const EdgeInsets.all(20),
                child: Text(
                  'Toplam: ${cart.totalPrice} TL',
                  style: const TextStyle(
                    fontSize: 20,
                    fontWeight: FontWeight.bold,
                  ),
                ),
              ),
            ],
          );
        },
      ),
    );
  }
}
```

To-Do Uygulaması (Bloc)

```
// Todo sınıfı
class Todo {
  final String id;
  final String task;
  bool complete;
```

```
  Todo({required this.id, required this.task, this.complete =
false});
}

// Events
abstract class TodoEvent {}

class AddTodo extends TodoEvent {
  final Todo todo;
  AddTodo(this.todo);
}

class ToggleTodo extends TodoEvent {
  final String id;
  ToggleTodo(this.id);
}

// States
class TodosState {
  final List<Todo> todos;
  TodosState(this.todos);
}

// Bloc
class TodosBloc extends Bloc<TodoEvent, TodosState> {
  TodosBloc() : super(TodosState([])) {
    on<AddTodo>((event, emit) {
      final updatedTodos =
List<Todo>.from(state.todos)..add(event.todo);
      emit(TodosState(updatedTodos));
    });

    on<ToggleTodo>((event, emit) {
      final updatedTodos = state.todos.map((todo) {
        if (todo.id == event.id) {
          return Todo(
            id: todo.id,
            task: todo.task,
            complete: !todo.complete,
          );
        }
        return todo;
      }).toList();
      emit(TodosState(updatedTodos));
```

```
    });
  }
}
```

Sonuç

Flutter’da durum yönetimi, uygulamanızın karmaşıklığına ve gereksinimlerine göre farklı yaklaşımlarla ele alınabilir. Basit uygulamalar için setState() veya Provider yeterli olabilirken, daha karmaşık uygulamalar için Bloc, Riverpod veya GetX gibi çözümler daha uygundur.

Doğru durum yönetimi yaklaşımını seçmek, Flutter uygulamanızın yapısını, bakımını ve ölçeklenebilirliğini doğrudan etkiler. Projenizin ihtiyaçlarını iyi analiz ederek, en uygun durum yönetimi stratejisini belirlemeniz önemlidir.

Unutmayın ki, durum yönetimi sadece bir araçtır ve amaç kullanıcılarınıza sorunsuz ve tutarlı bir deneyim sunmaktır.