Mobile Programming

Flutter State Management

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

setState | Provider | Riverpod | BLoC



Table of Contents

1. What is State Management?
2. Why State Management Matters
3. setState — The Built-in Approach
4. InheritedWidget — Flutter's Native DI
5. Provider — The Community Standard
6. Riverpod — Modern & Compile-Time Safe
7. BLoC / Cubit — Event-Driven Architecture
8. Choosing the Right Tool
9. Error & Loading State Patterns
10. Summary & Key Takeaways

What is State Management?

State is any data that can change over time and affects what the user sees.

Types of State

- **Ephemeral (Local):** UI-only state — scroll position, text input, toggle.
- **App (Global):** Shared data — user session, cart, theme, locale.
- **Business:** Domain data — products, orders, messages.

Where State Lives

- Inside a single widget (`setState`).
- Shared across widgets (Provider, Riverpod).
- Driven by events (BLoC).

> **Rule of Thumb:** If two or more widgets need the same data, lift the state up.

Why State Management Matters

Without proper state management, apps become:

- **Unpredictable:** UI doesn't reflect the latest data.
- **Untestable:** Business logic is tangled with widget code.
- **Unscalable:** Adding features breaks existing ones.

Goals of State Management:

1. **Separation of Concerns:** UI ≠ Business Logic.
2. **Testability:** Logic can be unit-tested without widgets.
3. **Performance:** Only rebuild widgets that actually changed.
4. **Predictability:** State transitions are explicit and traceable.

setState — The Built-in Approach

The simplest way to manage state. Flutter calls `build()` again after `setState()`.

```dart
class CounterPage extends StatefulWidget {
  const CounterPage({super.key});
  @override
  State<CounterPage> createState() => _CounterPageState();
}
class _CounterPageState extends State<CounterPage> {
  int _count = 0;
  void _increment() => setState(() => _count++);

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $_count'),
        ElevatedButton(onPressed: _increment, child: Text('Add')),
      ],
    );
  }
}
```

setState — When to Use & When Not To

Use `setState` when:

- State is **local** to a single widget.
- State is **simple** (a counter, a toggle, a text field).
- No other widget needs to know about it.

Avoid `setState` when:

- Multiple widgets need the same data.
- State involves async operations (API calls, DB queries).
- You need to persist state across page navigations.
- Business logic is mixed into the widget.

> **Anti-pattern:** A giant `setState` that rebuilds the entire page for a single toggle change.

setState - Step by Step Example

1. Create a StatefulWidget

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

setState - Step by Step Example

2. Create a variable inside the State class

```
class _MyPageState extends State<MyPage> {
  int counter = 0;
```

setState - Step by Step Example

3. Use the variable in the UI

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Text('$counter'),
    ),
```

4. Change the variable using `setState()`

```
floatingActionButton: FloatingActionButton(
  onPressed: () {
    setState(() {
      counter++;
    });
  },
),
```

setState

What `setState()` does

`setState()` tells Flutter:

> “Something changed. Rebuild the UI.”

InheritedWidget — Flutter's Native DI

Flutter's built-in mechanism for propagating data down the widget tree.

```
class InheritedCounter extends InheritedWidget {
  final int count;
  final VoidCallback increment;
  const InheritedCounter({
    super.key, required this.count,
    required this.increment, required super.child,
  });

  static InheritedCounter of(BuildContext context) {
    return context
      .dependOnInheritedWidgetOfExactType<InheritedCounter>()!;
  }

  @override
  bool updateShouldNotify(InheritedCounter old) => count != old.count;
}
```

InheritedWidget

Main Idea

```
Parent Widget
   ↓
InheritedWidget (shared data)
   ↓
Child Widgets use the data
```

Why use it?

- Share data efficiently
- Avoid prop drilling
- Updates dependent widgets automatically

InheritedWidget

Important

Libraries like `Provider` are built on top of `InheritedWidget`.

So usually:

- Small/simple → `setState()`
- Shared app state → `Provider`
- Low-level/manual control → `InheritedWidget`

Provider — The Community Standard

Provider wraps `InheritedWidget` with a clean API.

Core Concepts:

- `ChangeNotifier` — Notifies listeners when state changes.
- `ChangeNotifierProvider` — Provides the notifier to the widget tree.
- `context.watch<T>()` — Rebuilds when `T` changes.
- `context.read<T>()` — Accesses `T` without listening.

```
dependencies:
  provider: ^6.1.0
```

Provider — Implementation

```dart
class CounterModel extends ChangeNotifier { // 1. Model
  int _count = 0;
  int get count => _count;
  void increment() { _count++; notifyListeners(); }
}
void main() => runApp( // 2. Provider
  ChangeNotifierProvider(
    create: (_) => CounterModel(),
    child: const MyApp(),
  ),
);
// 3. Consume in a widget
class CounterText extends StatelessWidget {
  const CounterText({super.key});
  @override
  Widget build(BuildContext context) {
    final count = context.watch<CounterModel>().count;
    return Text('Count: $count');
  }
}
```

Provider — Multi-Provider & Selectors

Real apps have multiple state objects. Use `MultiProvider`.

```
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => UserModel()),
    ChangeNotifierProvider(create: (_) => CartModel()),
    ChangeNotifierProvider(create: (_) => ThemeModel()),
  ],
  child: const MyApp(),
)
```

Performance Tip: Use `Selector` to rebuild only when a specific field changes.

```
Selector<UserModel, String>(
  selector: (_, model) => model.name,
  builder: (_, name, __) => Text(name),
)
```

Provider - Step by Step Example

1. Add Provider

In `pubspec.yaml`:

```
dependencies:
  provider: ^6.0.0
```

Provider - Step by Step Example

2. Create a data class

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

class CounterProvider extends ChangeNotifier {
  int counter = 0;

  void increase() {
    counter++;
    notifyListeners();
  }
}
```

Provider - Step by Step Example

3. Provide the data

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => CounterProvider(),
      child: MyApp(),
    ),
  );
}
```

Provider - Step by Step Example

4. Read the data

```
Consumer<CounterProvider>(
  builder: (context, provider, child) {
    return Text('${provider.counter}');
  },
)
```

Provider - Step by Step Example

5. Update the data

```
ElevatedButton(
  onPressed: () {
    context.read<CounterProvider>().increase();
  },
  child: Text('Increase'),
)
```

Provider - Step by Step Example

Conclusion

When `increase()` runs:

```
notifyListeners();
```

All widgets using `CounterProvider` update automatically.

Riverpod — Modern & Compile-Time Safe

Riverpod is the **next generation** of Provider, by the same author (Remi Rousselet).

Key Improvements:

- **Compile-time safe** — No runtime `ProviderNotFoundException`.
- **No `BuildContext` required** — Use outside widgets.
- **Multiple providers of same type** — Unlike Provider.
- **Built-in async support** — `AsyncValue`, `FutureProvider`, `StreamProvider`.

```
dependencies:
  flutter_riverpod: ^2.5.0
```

Riverpod — Implementation

```dart
// 1. Define a provider
final counterProvider = StateProvider<int>((ref) => 0);

// 2. Wrap your app
void main() => runApp(const ProviderScope(child: MyApp()));

// 3. Consume in a widget
class CounterText extends ConsumerWidget {
  const CounterText({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final count = ref.watch(counterProvider);
    return Text('Count: $count');
  }
}

// 4. Modify state
ElevatedButton(
  onPressed: () => ref.read(counterProvider.notifier).state++,
  child: const Text('Add'),
)
```

Riverpod — AsyncNotifier Pattern

For real-world apps with API calls and async initialization.

```dart
@riverpod
class AsyncUsers extends _$AsyncUsers {
  @override
  Future<List<User>> build() async {
    final repo = ref.read(userRepositoryProvider);
    return repo.getAllUsers();
  }

  Future<void> refresh() async {
    state = const AsyncLoading();
    state = await AsyncValue.guard(() async {
      final repo = ref.read(userRepositoryProvider);
      return repo.getAllUsers();
    });
  }
}
```

Pattern: `AsyncValue.guard` wraps async calls and auto-handles loading/error/data.

BLoC / Cubit — Event-Driven Architecture

BLoC (Business Logic Component) separates UI from business logic using **events** and **states** via `Streams`.

Cubit is a simplified BLoC using **methods** instead of events.

- **Events** — Input from UI (`LoadUsers`, `AddToCart`).
- **States** — Output to UI (`UsersLoaded`, `UsersError`).
- **BLoC** — Transforms Events → States.
- **Cubit** — Methods emit states directly.

```
dependencies:
  flutter_bloc: ^8.1.0
```

Cubit — Simplified BLoC

```
// 1. State
sealed class CounterState {}
class CounterInitial extends CounterState {}
class CounterValue extends CounterState {
  final int count;
  CounterValue(this.count);
}

// 2. Cubit
class CounterCubit extends Cubit<CounterState> {
  CounterCubit() : super(CounterInitial());
  void increment() => emit(CounterValue(
    state is CounterValue ? (state as CounterValue).count + 1 : 1,
  ));
}

// 3. Provide
BlocProvider(create: (_) => CounterCubit(), child: CounterPage());
```

BLoC — Full Event-Driven Pattern

```dart
// Events
sealed class UserEvent {}
class LoadUsers extends UserEvent {}
// States
sealed class UserState {}
class UserLoading extends UserState {}
class UserLoaded extends UserState {
  final List<User> users;
  UserLoaded(this.users);
}
class UserError extends UserState {
  final String message;
  UserError(this.message);
}
// BLoC
class UserBloc extends Bloc<UserEvent, UserState> {
  final UserRepository repo;
  UserBloc(this.repo) : super(UserLoading()) {
    on<LoadUsers>((event, emit) async {
      emit(UserLoading());
      try {
        final users = await repo.getAllUsers();
        emit(UserLoaded(users));
      } catch (e) {emit(UserError(e.toString()));}
    });
  }
}
```

BLoC — UI Integration

```
BlocProvider(
  create: (_) => UserBloc(UserRepository())..add(LoadUsers()),
  child: Scaffold(
    body: BlocBuilder<UserBloc, UserState>(
      builder: (context, state) {
        if (state is UserLoading) return CircularProgressIndicator();
        if (state is UserError) return Text('Error: ${state.message}');
        if (state is UserLoaded) {
          return ListView.builder(
            itemCount: state.users.length,
            itemBuilder: (_, i) => ListTile(
              title: Text(state.users[i].name),
            ),
          );
        }
        return SizedBox.shrink();
      },
    ),
  ),
)
```

Choosing the Right Tool

Criteria	setState	Provider	Riverpod	BLoC/Cubit
Learning Curve	Trivial	Low	Medium	Medium-High
Scalability	Low	Medium	High	High
Testability	Low	Good	Excellent	Excellent
Boilerplate	Minimal	Low	Low-Medium	High
Best For	Prototypes	Small-Medium Apps	Medium-Large Apps	Enterprise Apps

Error & Loading State Patterns

Every async operation has **three** possible outcomes: **Loading**, **Data**, **Error**.

Riverpod Pattern

```
// AsyncValue handles all 3 states
class UserList extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final usersAsync = ref.watch(usersProvider);
    return usersAsync.when(
      loading: () => CircularProgressIndicator(),
      error: (e, _) => Text('Error: $e'),
      data: (users) => ListView(
        children: users.map((u) =>
          ListTile(title: Text(u.name))).toList(),
      ),
    );
  }
}
```

BLoC Pattern

```
// Explicit state matching
BlocBuilder<UserBloc, UserState>(
  builder: (context, state) {
    if (state is UserLoading) {
      return LoadingWidget();
    }
    if (state is UserError) {
      return ErrorWidget(state.message);
    }
    if (state is UserLoaded) {
      return UserListWidget(state.users);
    }
    return SizedBox.shrink();
  },
)
```

Summary: Key Takeaways

Core Concepts

- State = data that changes and affects UI.
- Separate UI from business logic.
- Three outcomes: Loading, Data, Error.

Tools

- `setState` — Local, simple state.
- `Provider` — Community standard, easy to learn.
- `Riverpod` — Modern, compile-time safe, recommended.

BLoC/Cubit

- Event-driven, highly testable.
- Best for large teams and enterprise apps.

Best Practices

- Keep state close to where it's used.
- Use immutable state objects.
- Handle all three async states.
- Use `Selector` / `buildWhen` for performance.

Thank You! 🙌

Questions?

"Why did the Flutter developer go broke? Because he used setState for everything and couldn't afford the rebuilds."