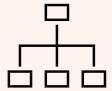
CROSS-PLATFORM MOBILE APP DEVELOPMENT

MIDTERM

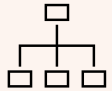
522K0031 – Phạm Thái An Bình

522K0036 – Trần Hoàng Hiếu

Topic  State Management in Flutter

What We Built  An Expense Tracker App

State management  Riverpod

Topic  State Management in Flutter

What We Built  An Expense Tracker App

State management  Riverpod

---

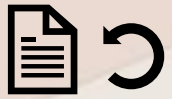
☆ Key features Add/track expenses

Category management

Real-time calculations

Cross-platform persistence

Why State Management Matters



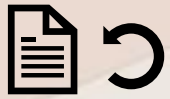
What is State?

Any data that your application needs to remember and that can change over time.

What is State Management?

The process of managing the data that your application uses.

Why State Management Matters



What is State?

Any data that your application needs to remember and that can change over time.

What is State Management?

The process of managing the data that your application uses.

Why is it crucial in Flutter?

What is Riverpod?

A compile-safe, Provider-based state management solution for Flutter.

Key Advantages:

- Compile-safe
- Decoupled from the Widget Tree
- Declarative and Reactive
- Scalable and Testable
- Flexible

Riverpod: Core Concept 1: The Provider

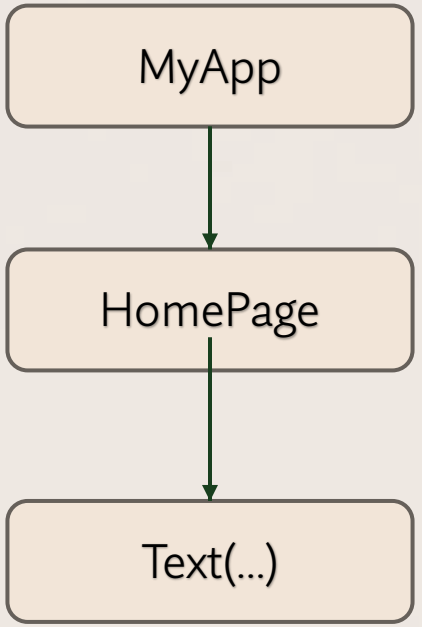
Definitions

```
final greetingProvider =
Provider((ref)
 =>return "Hello!");
```

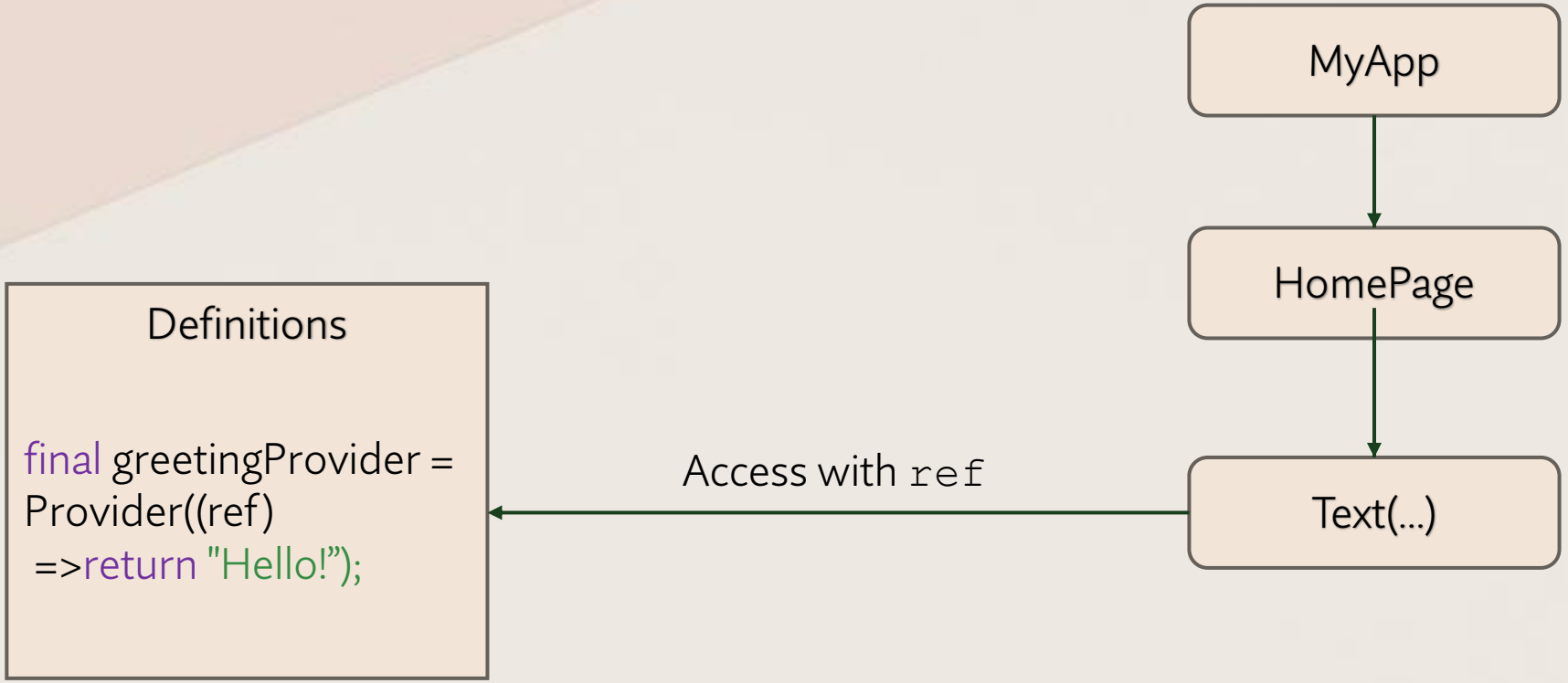
Riverpod: Core Concept 1: The Provider

Definitions

```
final greetingProvider =
Provider((ref)
 =>return "Hello!");
```



Riverpod: Core Concept 1: The Provider

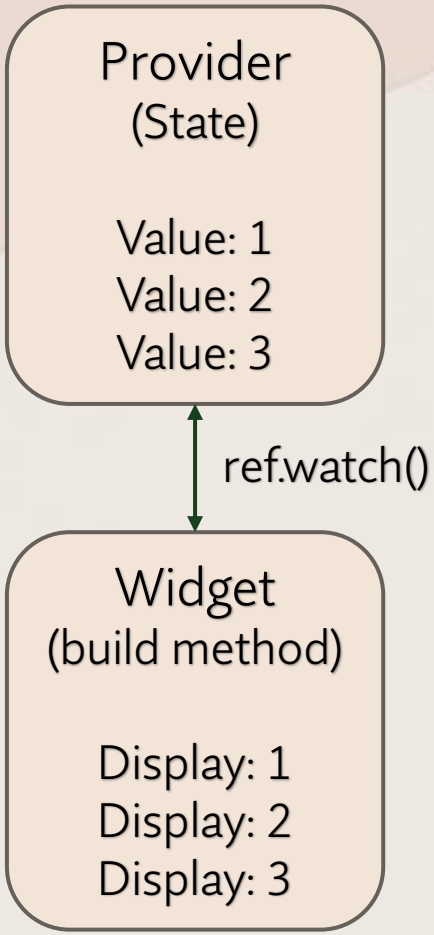


Riverpod: Core Concept 2: The `ref`

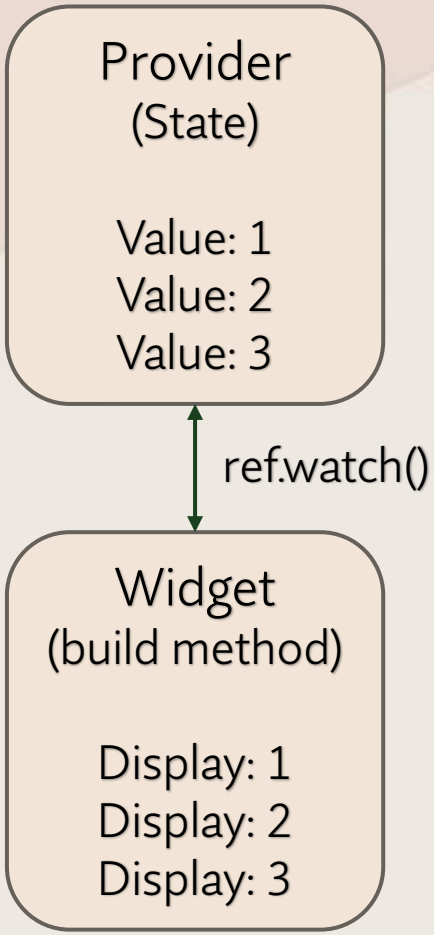
```
// A widget must be a “ConsumerWidget” to get the ‘ref’
class MyTextWidget extends ConsumerWidget {

    @override
    Widget build(BuildContext context, WidgetRef ref) {

        return Text(...);
    }
}
```

`ref`: Your widget's "key card" for accessing providers.

Riverpod: Core Concept 3: ref.watch vs ref.read

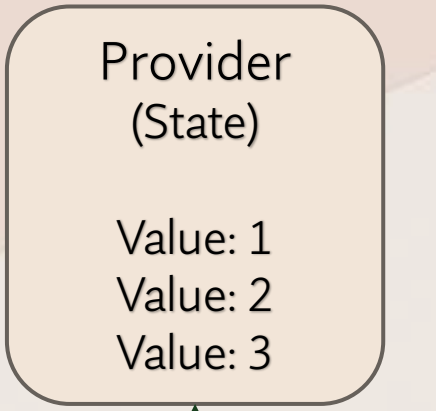


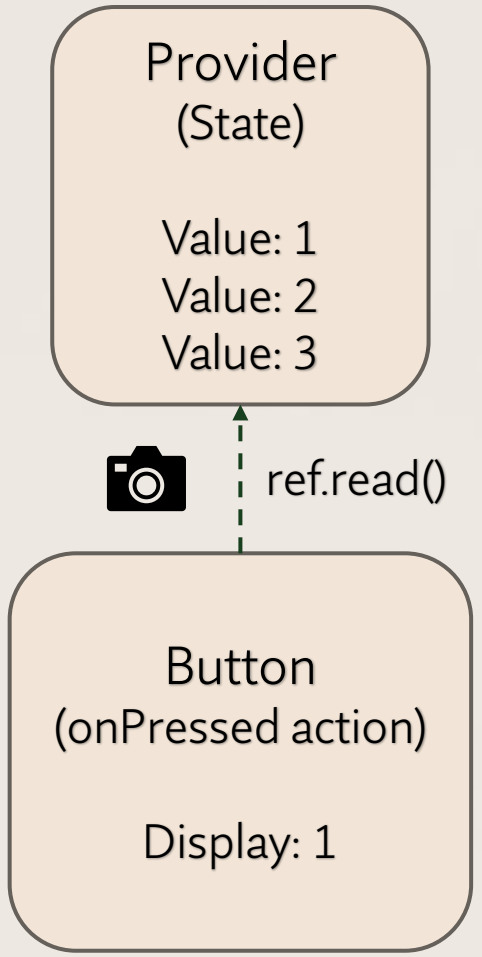
Riverpod: Core Concept 3: ref.watch vs ref.read



Result: The Widget is *always* in sync.
It rebuilds on *every* state change

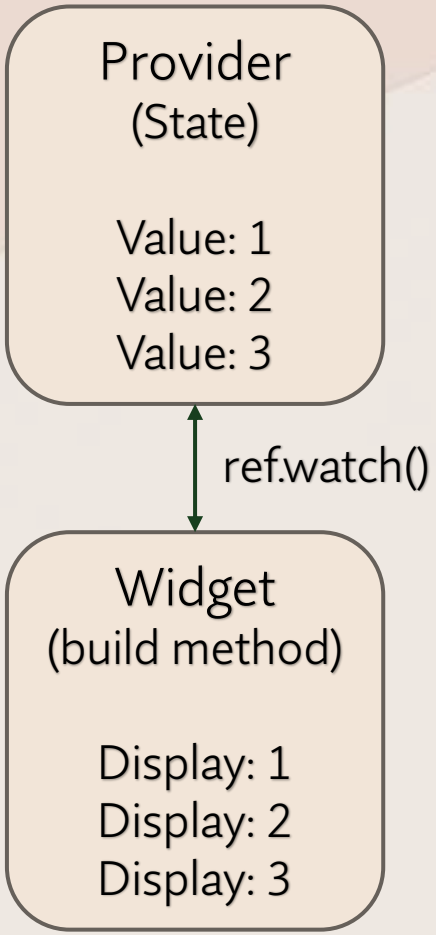
Riverpod: Core Concept 3: ref.watch vs ref.read



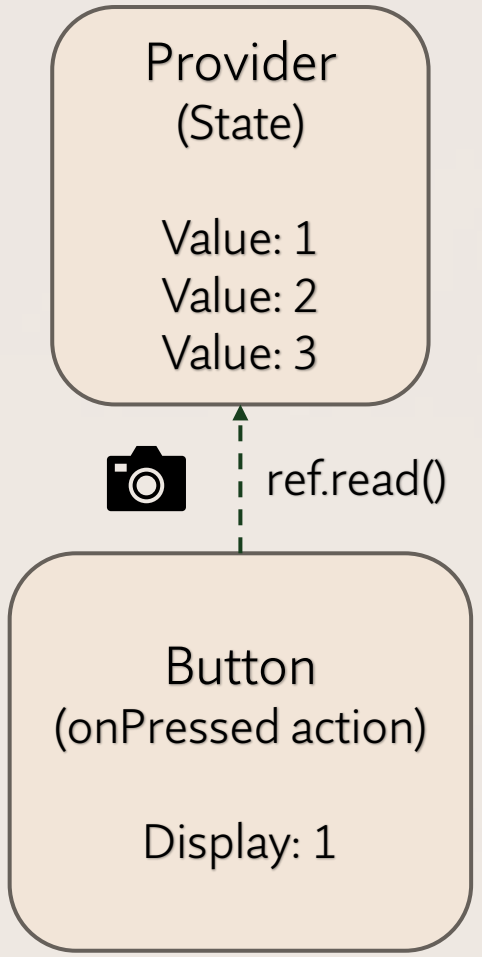


Result: The Widget is *always* in sync.
It rebuilds on *every* state change

Riverpod: Core Concept 3: ref.watch vs ref.read

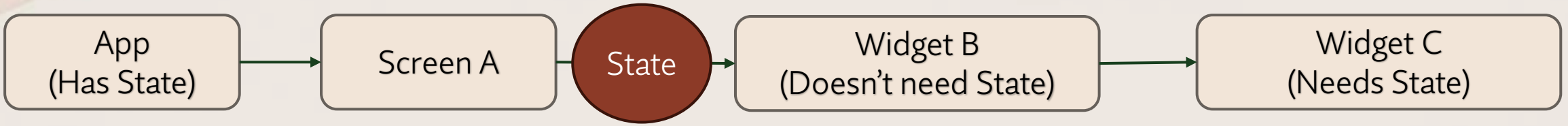


Result: The Widget is *always* in sync.
It rebuilds on *every* state change



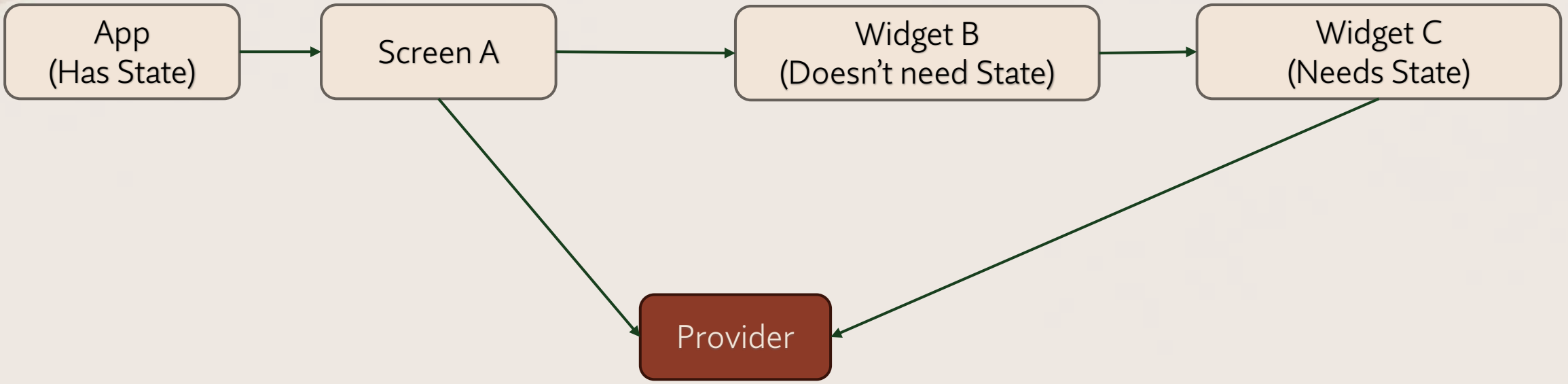
Result: The Button gets the state *once* (at Value: 1). It is *unaware* of later changes and does *not* trigger a rebuild.

Riverpod: Core Architecture & Principles (1/2)

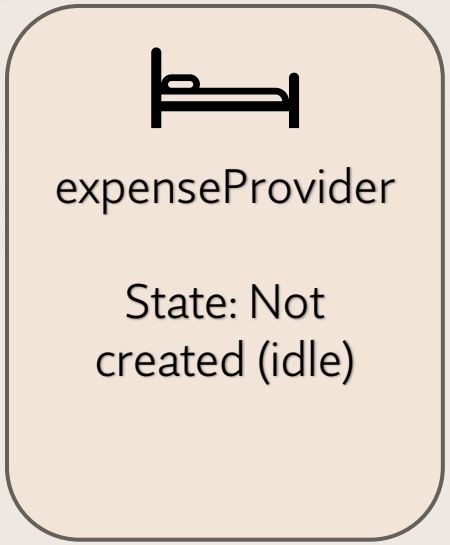


Data needs to get to Widget C, but has to go through Widget B first.

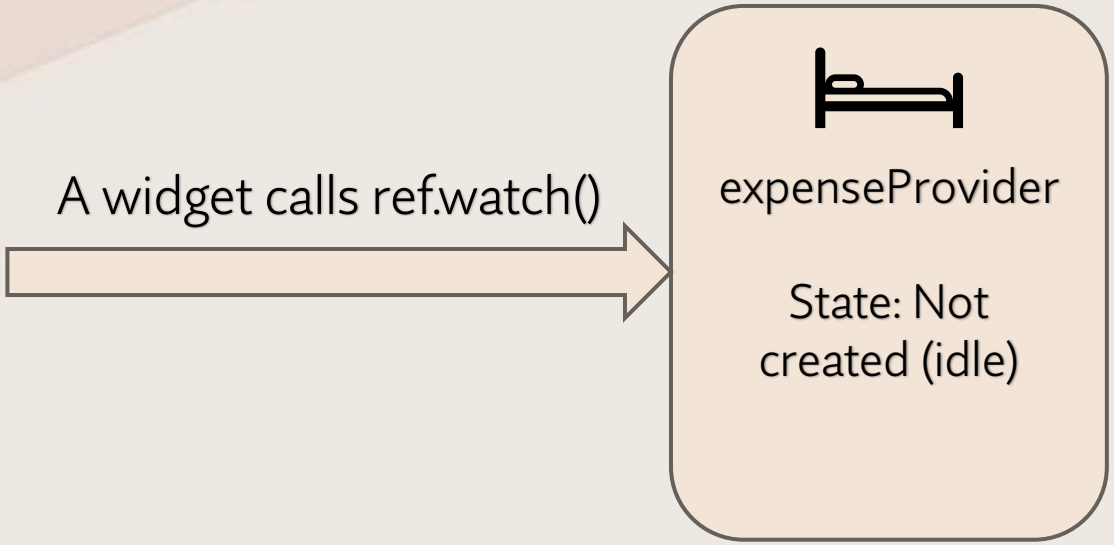
Riverpod: Core Architecture & Principles (1/2)



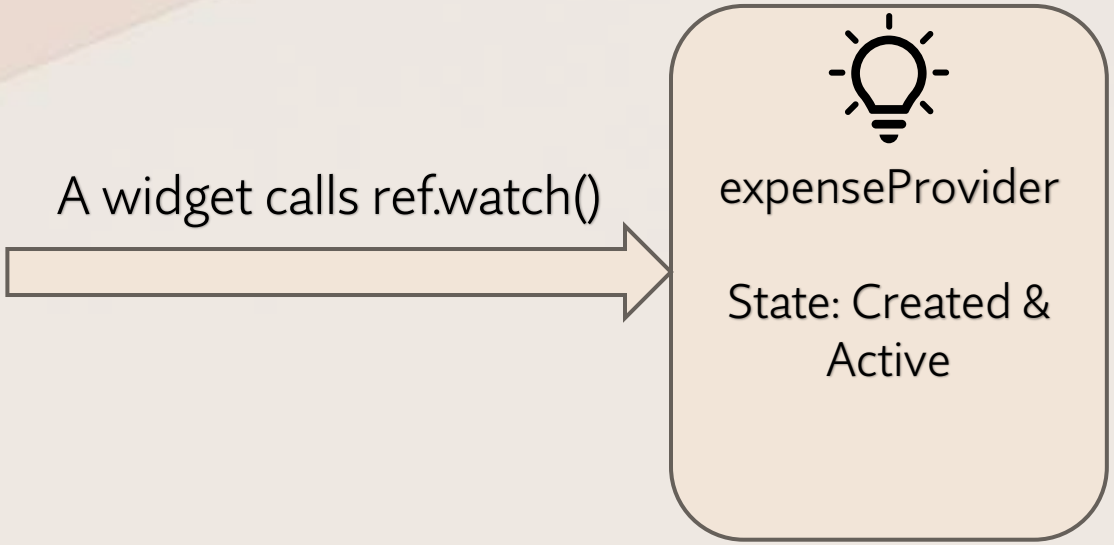
Riverpod: Core Architecture & Principles (1/2)



Riverpod: Core Architecture & Principles (1/2)

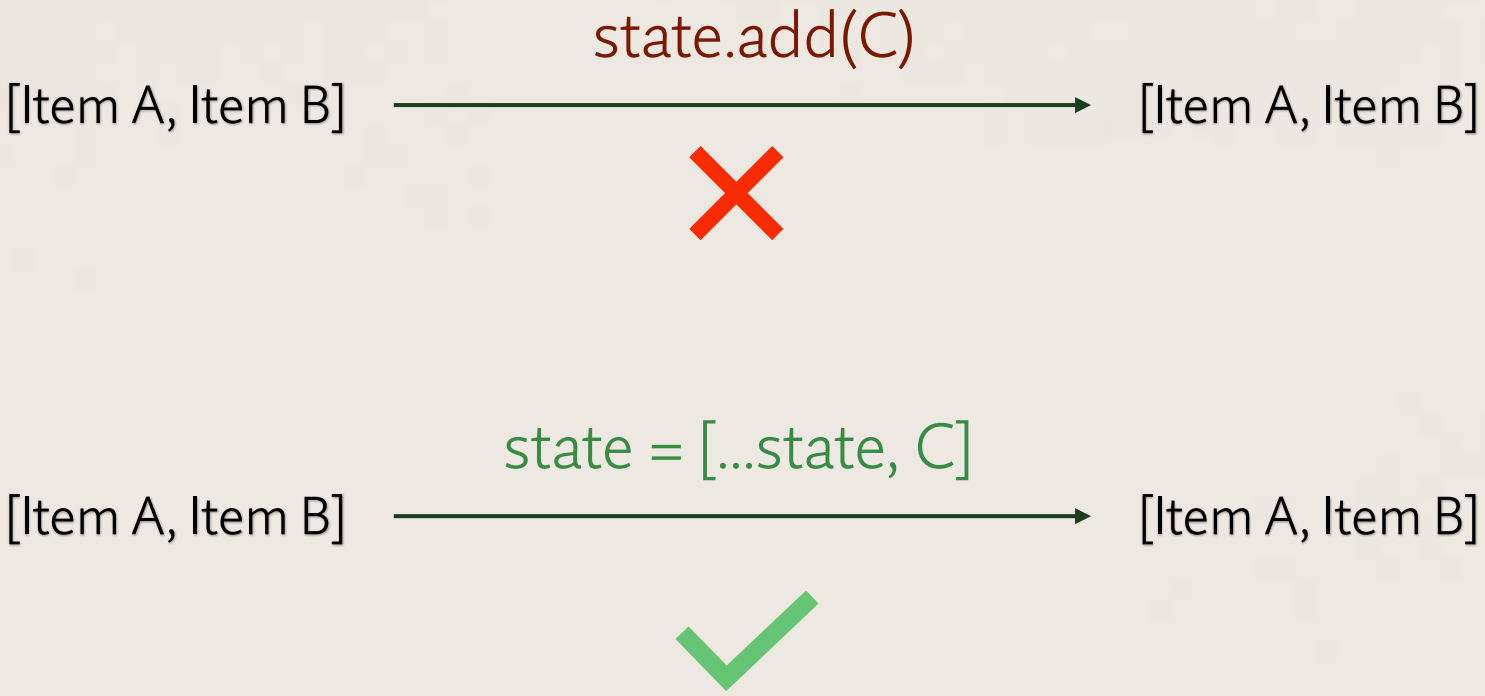


Riverpod: Core Architecture & Principles (1/2)



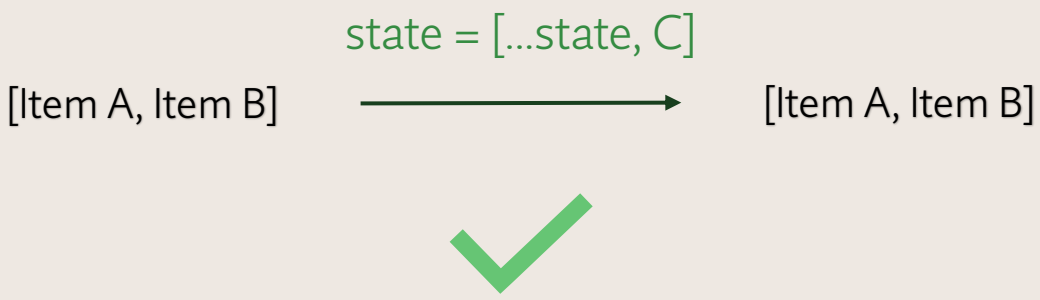
Riverpod: Core Architecture & Principles (2/2)

Immutable State



Riverpod: Core Architecture & Principles (2/2)

Immutable State

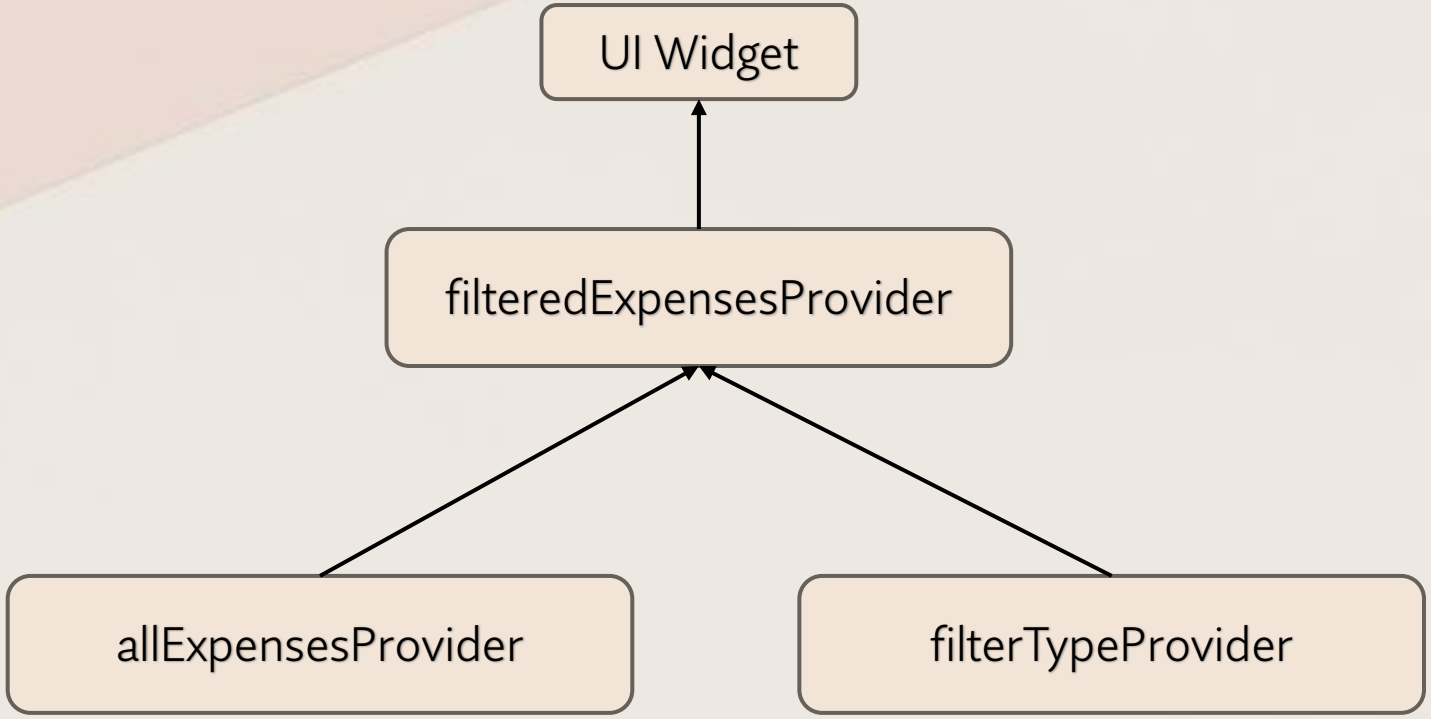


Compile-safe

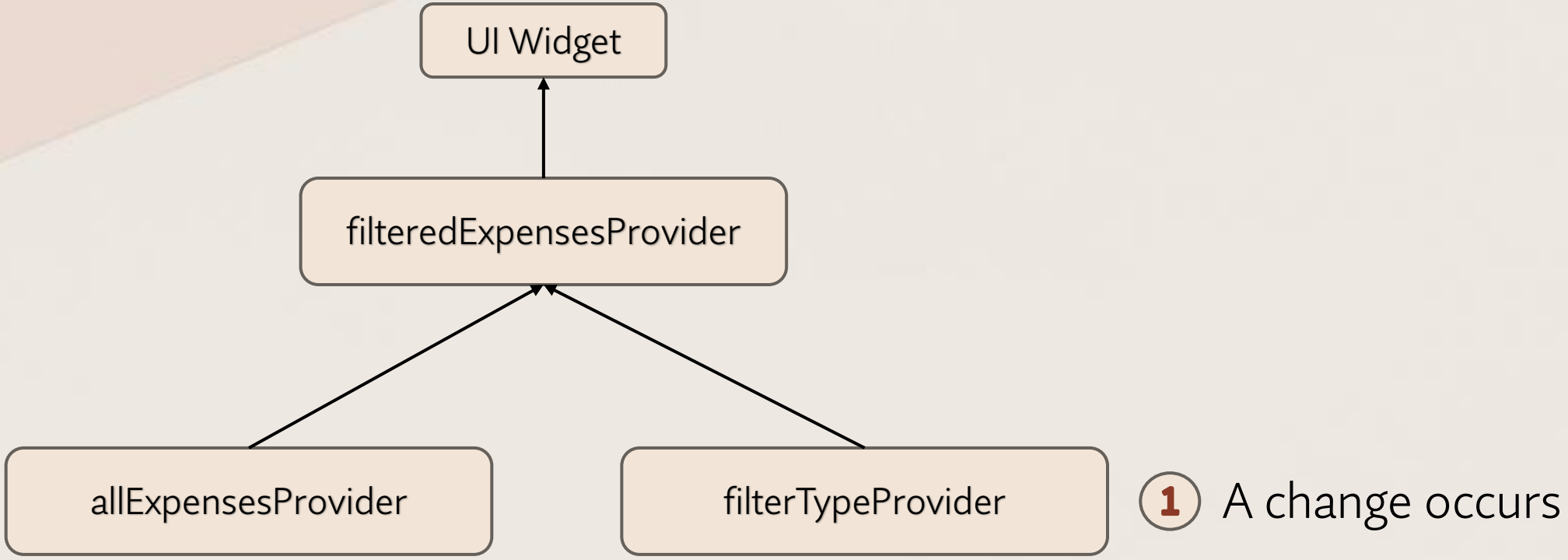
```
ref.watch(myPrvoider);
```

Riverpod is **compile-safe**, so typos in provider names are caught as errors *before* the app runs, which prevents them from causing **runtime crashes** for the user.

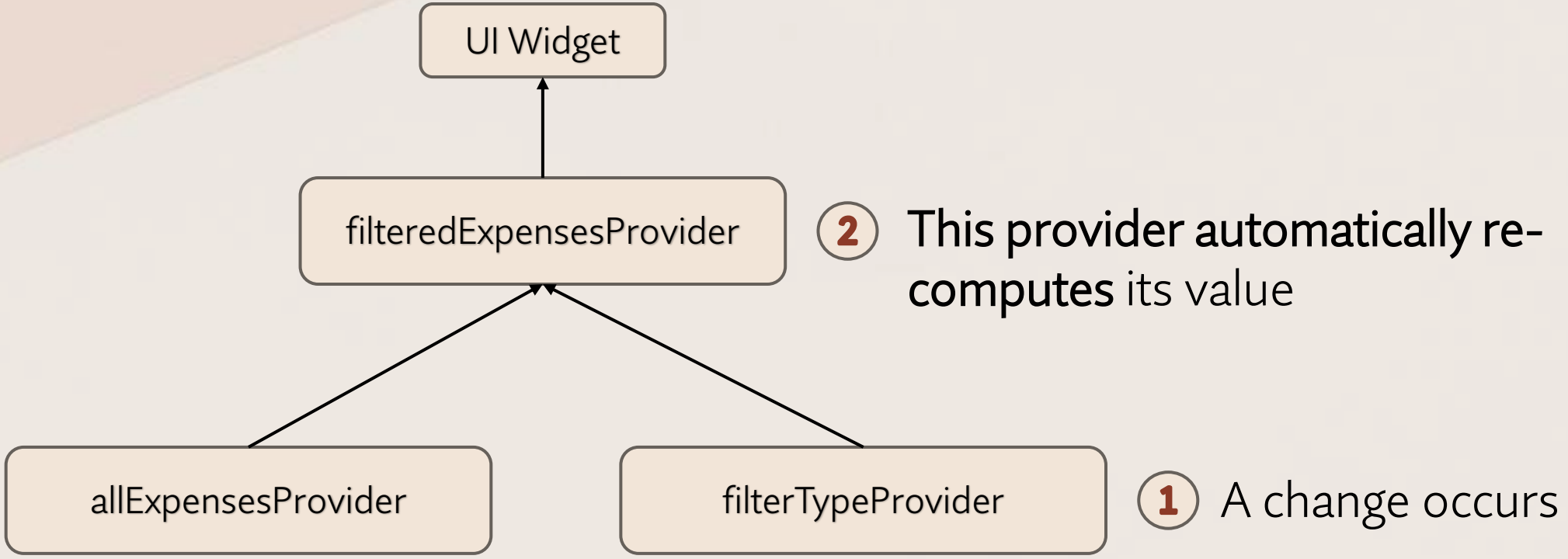
Architecture: Combining Providers



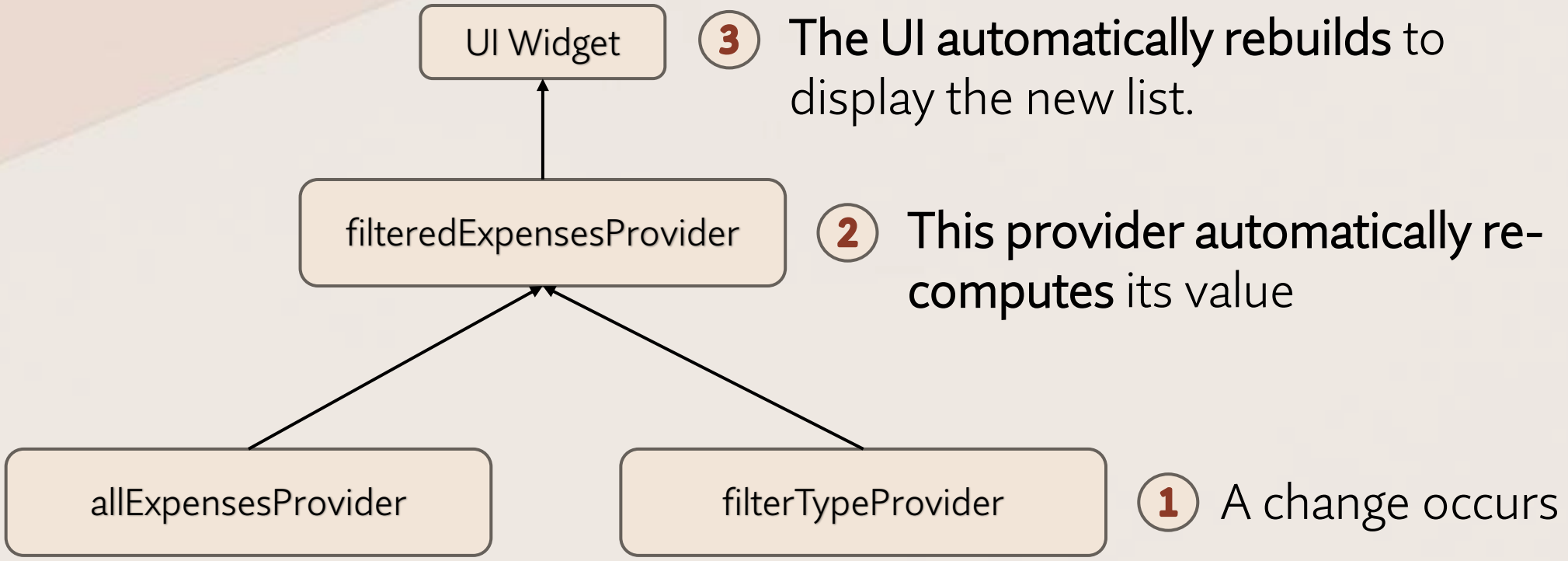
Architecture: Combining Providers



Architecture: Combining Providers



Architecture: Combining Providers



The UI layer remains simple. It only watches *one* provider and rebuilds automatically. The logic is encapsulated and decoupled.

Riverpod: Provider types

Provider

- Read-only data
- totalExpense, categories

StateProvider

- Simple mutable state
- Selected filter, theme

StateNotifierProvider

- Complex state with logic
- ExpenseList, budget

FutureProvider

- Async data loading
- loadExpensesFromDBProvider

StreamProvider

- Real-time data streams
- Expense changes, expenseUpdatesStreamProvider

Live Demo

Code Comparison: How State is *Changed*

Counter Notifier

```
void increment() {

    state = state + 1;
}
```

State is **replaced**.

CounterCutbit

```
void increment() {

     emit(state + 1);
}
```

State is **replaced**.

CounterStore

```
@action
void increment() {

     value++;
}
```

State is **changed in-place**.

Code Comparison: How State is *Read*

Riverpod

Counter Notifier

```
@override
Widget build(BuildContext
context, WidgetRef ref) {
  final count =
ref.watch(counterProvider);
  return Text('$count');
}
```

Code Comparison: How State is *Read*

Riverpod

Counter Notifier

```
@override
Widget build(BuildContext
context, WidgetRef ref) {
  final count =
ref.watch(counterProvider);
  return Text('$count');
}
```

BLoC

CounterCutbit

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

Code Comparison: How State is *Read*

Riverpod

Counter Notifier

```
@override
Widget build(BuildContext
context, WidgetRef ref) {
  final count =
ref.watch(counterProvider);
  return Text('$count');
}
```

BLoC

CounterCutbit

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

MobX

CounterStore

```
Observer(
  builder: (_) {
    return
Text('${counter.value}');
  },
)
```

Code Comparison: How State is *Read*

Riverpod

Counter Notifier

```
@override
Widget build(BuildContext
context, WidgetRef ref) {
  final count =
ref.watch(counterProvider);
  return Text('$count');
}
```

BLoC

CounterCutbit

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

MobX

CounterStore

```
Observer(
  builder: (_) {
    return
Text('${counter.value}');
  },
)
```

Riverpod and BLoC are **explicit**, requiring you to use commands like `ref.watch()` or `BlocBuilder` to listen to state. Conversely, MobX is **implicit**; its `Observer` widget *automatically* detects and subscribes to any observables used inside it.

Code Comparison: How Async State is Handled

Riverpod

Counter Notifier

```
ref.watch(myProvider).when(
  data: (data) => ...,
  loading: () => Spinner(),
  error: (e, s) => Error(),
)
```

Built-in state object

Code Comparison: How Async State is Handled

Riverpod

Counter Notifier

```
ref.watch(myProvider).when(
  data: (data) => ...,
  loading: () => Spinner(),
  error: (e, s) => Error(),
)
```

Built-in state object

BLoC

CounterCutbit

```
BlocBuilder<DataBloc, DataState>(
  builder: (context, state) {
    if (state is DataLoading) ...
    if (state is DataError) ...
    if (state is DataSuccess) ...
  },
)
```

Manual, developer-defined state classes

Code Comparison: How Async State is Handled

Riverpod

Counter Notifier

```
ref.watch(myProvider).when(
  data: (data) => ...,
  loading: () => Spinner(),
  error: (e, s) => Error(),
)
```

Built-in state object

BLoC

CounterCubit

```
BlocBuilder<DataBloc, DataState>(
  builder: (context, state) {
    if (state is DataLoading) ...
    if (state is DataError) ...
    if (state is DataSuccess) ...
  },
)
```

Manual, developer-defined state classes

MobX

CounterStore

```
Observer(builder: (_) {
  if (store.isLoading) ...
  if (store.error != null) ...
  if (store.data != null) ...
})
```

Manual, 'DIY' state variables

Code Comparison: How Async State is Handled

Riverpod

Counter Notifier

```
ref.watch(myProvider).when(
  data: (data) => ...,
  loading: () => Spinner(),
  error: (e, s) => Error(),
)
```

Built-in state object

BLoC

CounterCutbit

```
BlocBuilder<DataBloc, DataState>(
  builder: (context, state) {
    if (state is DataLoading) ...
    if (state is DataError) ...
    if (state is DataSuccess) ...
  },
)
```

Manual, developer-defined state classes

MobX

CounterStore

```
Observer(builder: (_) {
  if (store.isLoading) ...
  if (store.error != null) ...
  if (store.data != null) ...
})
```

Manual, 'DIY' state variables

Riverpod provides a **built-in** solution (AsyncValue/.when) that automatically handles loading/error states. In contrast, BLoC requires **manual but structured** state classes (Loading, Success, Error), while MobX needs **manual "DIY"** variables managed within the store.

Summary

Feature	Riverpod	BLoC (Cubit)	MobX
Core Paradigm	**Immutable**	**Immutable**	**Mutable**
UI Reactivity	**Explicit** (ref.watch)	**Explicit** (context.watch)	**Implicit** (Observer)
Async Handling	**Built-in** (.when)	**Manual** (Structured States)	**Manual** (DIY Variables)
Debugging	**Compile-Safe** (Explicit)	**Compile-Safe** (Explicit)	**“Quiet error"** (Implicit)

Thank you