

Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 3
State Management

Lecture 1



Chapter
01

Table Of Content

- What is State Management?
- State Management over Stateful widget
- What is Riverpod?
- Provider Scope
- Types of Provider



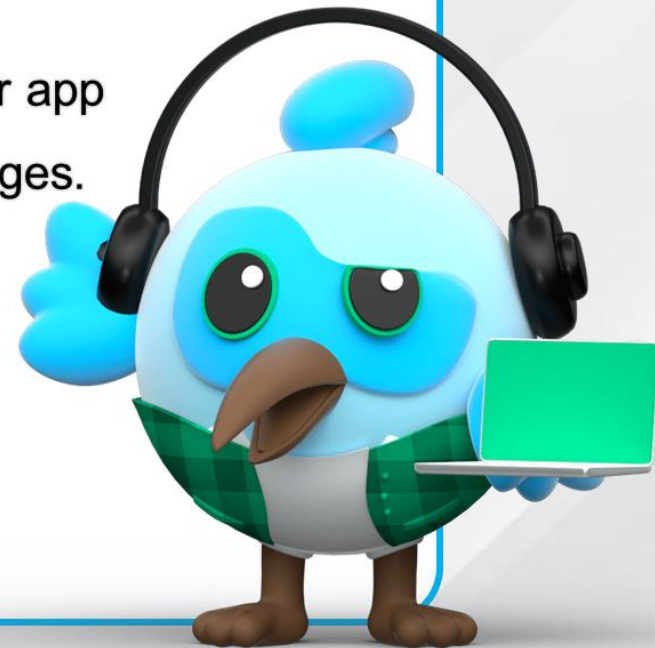
Introduction

Introduction

What is State Management?

State Management is:

The process of handling data that changes over time in your app and making sure your UI updates correctly when that data changes.



Think of "State" as:

- User input in a form
- The current page or tab
- Whether the user is logged in
- Items in a shopping cart
- API-loaded data like a list of products

Without State Management:

- Your app UI won't know when the data changes.
- You'll write **messy**, **repetitive**, or **hard-to-maintain** code.

With State Management:

- Data is stored in a **centralized and predictable way.**
- The UI **reacts** to changes **automatically.**
- Makes your code **modular, testable, scalable.**

In Flutter, common state management approaches:

Tool	Description
<code>setState</code>	Built-in, for local/small UI states
<code>Provider</code>	Google-recommended, good for shared/global state
<code>Riverpod</code>	Advanced, type-safe, and testable (replacement for Provider)
<code>Bloc/Cubit</code>	Business logic-focused, popular in complex apps
<code>GetX, MobX, etc.</code>	Other third-party solutions

State Management over Stateful widget

When **Stateful Widget** is OK

Use it for **very small, local UI changes**, like:

- A toggle button
- A single counter
- Simple animations

```
class MyCounter extends StatefulWidget {  
  @override  
  _MyCounterState createState() => _MyCounterState();  
}  
  
class _MyCounterState extends State<MyCounter> {  
  int count = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    return ElevatedButton(  
      onPressed: () => setState(() => count++),  
      child: Text('Count: $count'),  
    );  
  }  
}
```

Why **Stateful Widget** Fails in Bigger Apps

Problem	Explanation
 Hard to Share State	You can't access the state outside the widget easily.
 Unnecessary UI Rebuilds	<code>setState</code> rebuilds the whole widget unnecessarily.
 Tightly Coupled Code	Business logic and UI are mixed together.
 Hard to Test	You can't test state logic without running the UI.
 Nested Widgets Hell	Deep widget trees make it painful to pass state or callbacks.
 Loses State on Navigation	Unless lifted, state disappears when widget is removed from the tree.

Why Use a State Management Solution like Riverpod

Benefit	Riverpod Helps You...
 Share state globally	Define state once, use it anywhere in your app
 Decouple UI & logic	Keep business logic in providers or notifiers
 Test easily	Test logic without needing the UI
 Reactive UI	UI updates automatically when state changes
 Clean architecture	Organize your code into layers (data, logic, UI)
 Modular & reusable	Reuse providers across screens and features

What is Riverpod

Why Riverpod?

- **Modern and robust alternative to Provider**
- Compile-time safety & IDE support
- Works with **dart** (not just Flutter widgets)
- Supports modular architecture

Key Concepts in Riverpod

- **Provider** – a basic way to expose state
- **Consumer/WidgetRef** – reads provider values
- **StateProvider** – simple mutable state
- **FutureProvider / StreamProvider** – async data
- **StateNotifier / StateNotifierProvider** – advanced state control
- **AsyncValue** – handles loading, error, data

Provider Scope

What is **ProviderScope** in Riverpod?

ProviderScope is the **root widget** that stores and manages all of your Riverpod providers.

✅ Definition:

```
void main() {  
  runApp(  
    ProviderScope( // 👉 this wraps your whole app  
      child: MyApp(),  
    ),  
  );  
}
```

Think of it as:

 A container that **holds the state** of your providers and makes them accessible to all widgets below it in the widget tree.

Why is it required?

- Without **ProviderScope**, you **cannot use any provider** (like **ref.watch(...)**) because there's no place for Riverpod to **store state**.
- It also handles **overrides, caching, resource disposal**, and **provider dependencies**.

Features of `ProviderScope`:

Feature	Description
 Stores state	Keeps track of each provider's state
 Enables <code>ref.watch</code>	Allows widgets to listen to providers
 Testing support	Makes it easy to test with <code>ProviderContainer</code>
 Overrides	Allows you to override a provider (e.g., for testing or environment config)

Summary:

-  **ProviderScope** is **required** to use Riverpod.
-  It holds all the **provider states**.
-  Useful for **testing, overriding, and managing scope** of state.

Types of Provider

Provider

► **Purpose:** Exposes read-only immutable values (e.g., configuration, services).

✅ **Use** when you don't expect the value to change during runtime.

🧠 **Example:**

```
final configProvider = Provider<String>((ref) => 'production');
```

```
final config = ref.watch(configProvider);
```

"Mutable" (not *mutable*) means:

A value or object **can change** over time after it's created.

Mutable = Changeable

```
var x = 5;  
x = 10; //  This is mutable because we changed the value
```

Immutable = Not Changeable

```
const y = 5;  
y = 10; // ❌ Error! This is immutable.
```

In Riverpod terms:

Provider Type	Mutable?	Meaning
Provider	✗ No	The value is fixed (read-only)
StateProvider	☒ Yes	You can update <code>.state</code> to change the value
State Notifier	☒ Yes	You define methods to mutate state (<code>state = newValue</code>)
Future Provider	✗ No	Once the future resolves, the value is fixed
StreamProvider	☒ Yes	It updates based on stream events (like chat messages)

StateProvider<T>

► **Purpose:** Holds a mutable state of type **T** (int, String, bool, etc.)

✅ **Great for simple local state like toggles, counters, inputs.**

🧠 **Example:**

```
final counterProvider = StateProvider<int>((ref) => 0);
```

```
// Read  
final count = ref.watch(counterProvider);  
// Write  
ref.read(counterProvider.notifier).state++;
```

FutureProvider<T>

► **Purpose:** Handles asynchronous computations returning a Future.

✅ **Use for loading data from APIs, files, or async tasks.**

🧠 **Example:**

```
final userProvider = FutureProvider<User>((ref) async {  
    return await fetchUser();  
});  
  
final userAsync = ref.watch(userProvider);  
userAsync.when(  
    data: (user) => Text(user.name),  
    loading: () => CircularProgressIndicator(),  
    error: (e, _) => Text('Error: $e'),  
);
```

StreamProvider<T>

► **Purpose:** Exposes stream-based asynchronous data.

✅ **Use for real-time updates like Firebase snapshots or WebSockets.**

🧠 **Example:**

```
final chatStreamProvider = StreamProvider<List<Message>>((ref) {  
    return chatRepository.messageStream();  
});  
  
ref.watch(chatStreamProvider).when(...);
```

StateNotifier<T>

► **Purpose:** Encapsulates complex state logic in a class.

✓ **Use when state has logic like counters, API calls, or toggling.**

🧠 **Example:**

```
class CounterNotifier extends StateNotifier<int> {  
    CounterNotifier() : super(0);  
    void increment() => state++;  
}  
  
final counterNotifierProvider =  
    StateNotifierProvider<CounterNotifier, int>((ref) => CounterNotifier());  
  
ref.read(counterNotifierProvider.notifier).increment();
```

ConsumerWidget, ConsumerStatefulWidget

► **Purpose:** Special widgets that let you `ref.watch()` inside the build method.

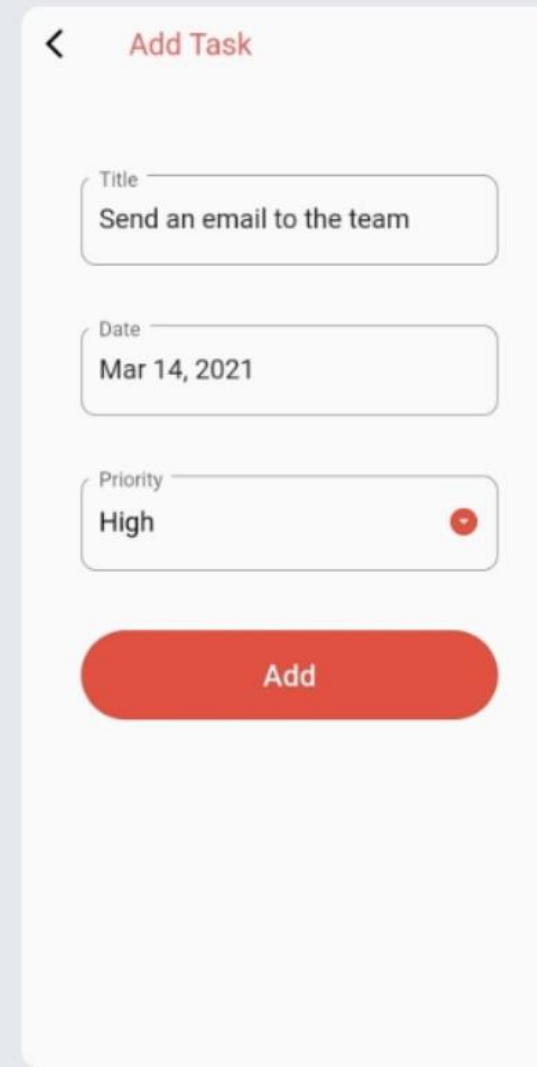
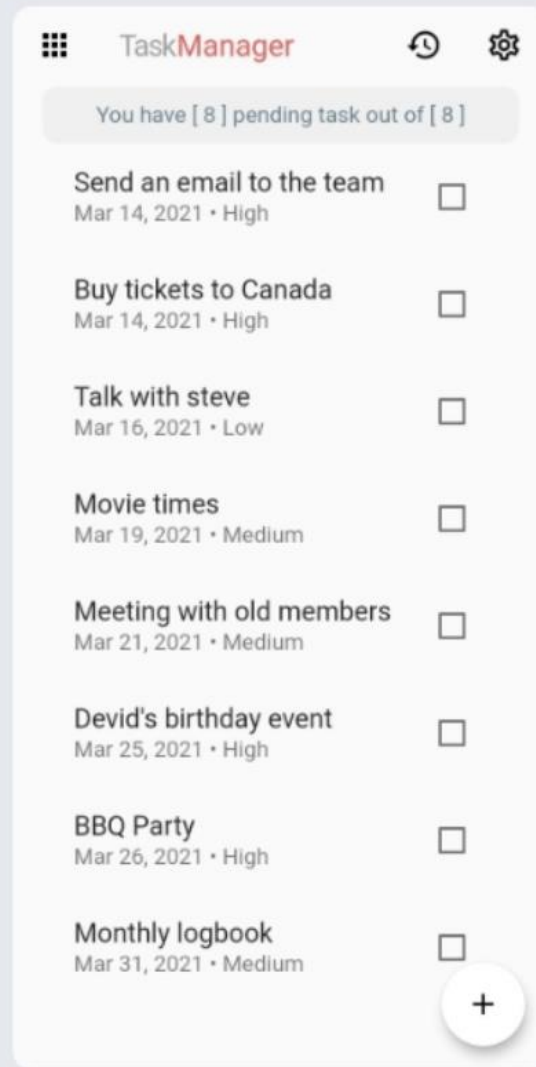
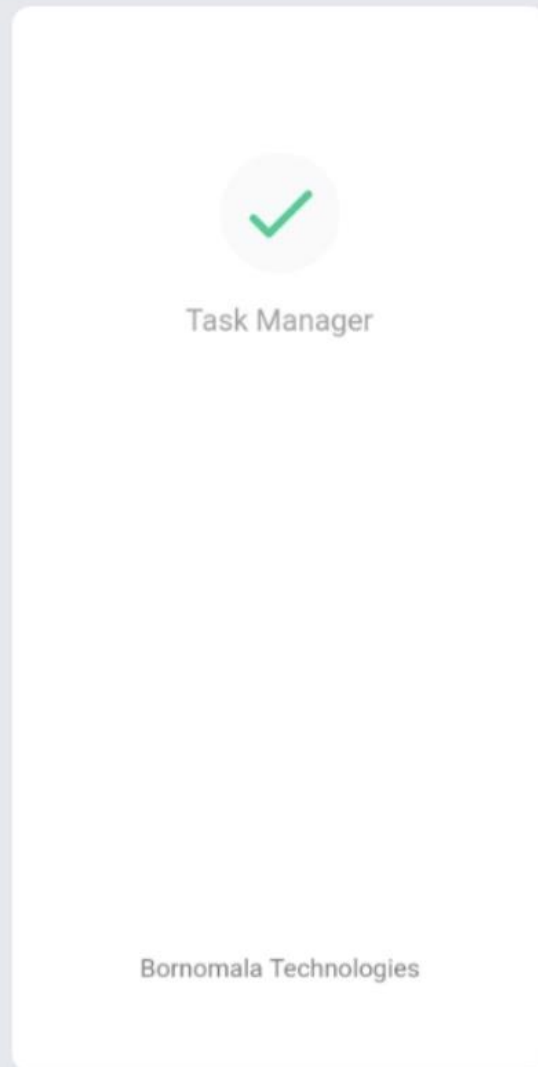
```
class MyWidget extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    final count = ref.watch(counterProvider);  
    return Text('$count');  
  }  
}
```

`ref.read`, `ref.watch`,
`ref.listen`

Method	Use Case
<code>ref.read</code>	Access value once or write/mutate
<code>ref.watch</code>	Rebuilds widget when provider value changes
<code>ref.listen</code>	Executes a callback when provider changes

Type	Use For	Mutable	Async	UI Aware
Provider	Config/Services	✗	✗	✗
StateProvider	Simple local state	✓	✗	✓
FutureProvider	Async Future data	✗	✓	✓
StreamProvider	Async Stream data	✗	✓	✓
StateNotifier	Advanced state logic	✓	✗	✗
StateNotifierProvider	Binding StateNotifier to state	✓	✗	✓
Notifier	Riverpod 2.x logic class	✓	✗	✓

What is Next ?



< Update Task

Title

BBQ Party with friends

Date

Mar 26, 2021

Priority

High



Update

Delete

< History

You have completed [1] tasks

Movie times

Mar 19, 2021 • Medium



Task reassigned



Thank You