Provider

State Management: ChangeNotifier Fundamentals

Learning Objectives

By the end of this lecture, you will be able to:

- Understand what **state** means in Flutter applications
- Explain the **ChangeNotifier** pattern
- Implement basic state management using **Provider**
- Differentiate between **Consumer** and **context.read()**
- Prepare for **MVVM architecture** patterns

What is State?

State = Data that can change over time and affects the UI

Examples of State:

- User login status
- Shopping cart items
- Form input values
- Loading indicators
- Counter values

Problem: How do we manage state across multiple widgets?

Solution

For shared or app-wide state → We can use **state management tools**: **Provider, Riverpod, BLoC, Redux**

We Choose Provider State Management Tools

1. Official & Built-in

- Developed by the **Flutter team**
- Integrated into Flutter ecosystem
- Actively maintained & documented

2. Simple & Lightweight

- Easy to learn, minimal boilerplate
- Works directly with Flutter's widget tree
- Extends existing `InheritedWidget` mechanism

3. Scales with App

- Handles **small local state** or **app-wide state**
- Plays well with **ChangeNotifier** for reactivity
- Supports dependency injection

Flutter State Management Evolution

We have used Stateful Widget/States for updating screen, but from now on, we use ChangeNotifier in the Provider State Management tool.

```
StatefulWidget → ChangeNotifier → Advanced Solutions
     ↓               ↓                    ↓
   Local State   Shared State       Complex State
   Simple Apps   Medium Apps       Large Apps
```

Today's Focus: ChangeNotifier for shared state

The Problem with StatefulWidget

```dart
class CounterWidget extends StatefulWidget {
  @override
  _CounterWidgetState createState() => _CounterWidgetState();
}

class _CounterWidgetState extends State<CounterWidget> {
  int _counter = 0;  // State is locked inside this widget

  void _increment() {
    setState(() { _counter++; });
  }
}
```

Issue: State is trapped in one widget! 🔒

ChangeNotifier to the Rescue!

ChangeNotifier is a simple class that provides change notifications

Key Features:

- ✅ Separate state from UI
- ✅ Share state across multiple widgets
- ✅ Notify listeners when state changes
- ✅ Foundation for MVVM pattern

ChangeNotifier Basics

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

class CounterModel extends ChangeNotifier {
  int _count = 0;                       // Private state

  int get count => _count;              // Public getter

  void increment() {
    _count++;
    notifyListeners();                  // 🔑 Key method!
  }
}
```

Rule: Always call `notifyListeners()` when state changes!

We used setState(() => increment()).

The Provider Package

Provider connects ChangeNotifier to the widget tree

```
dependencies:
  provider: ^6.1.1
```

We can use Provider in `dartpad.dev`, but we should add dependency when we use Provider in the Flutter application.

Three main components:

1. **ChangeNotifierProvider** - Creates and provides the model
2. **Consumer** - Listens and rebuilds when state changes
3. **context.read()** - Accesses model without listening

Setting Up Provider

We use `ChangeNotifierProvider` class with the Model that inherits from ChangeNotifier as an argument.

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CounterModel(),  // Create model
      child: MyApp(),
    ),
  );
}
```

Now `CounterModel` is available to all child widgets!

Consuming State: Consumer Widget

We have used a builder method in the Statefule Widget to redraw widgets, but in this state management model, we use `Consurem<T>` class to redraw widgets.

```
Consumer<CounterModel>(
  builder: (context, counterModel, child) {
    return Text(
      '${counterModel.count}',          // Access state
      style: TextStyle(fontSize: 48),
    );
  },
)
```

The builder in the Consumer<CounterModel> automatically rebuilds when `notifyListeners()` is called
We used the builder method in the State class.

Triggering Actions: context.read()

We have used setState() method with a lambda expression to trigger actions, but in this state management model, we use `context.read<T>()` to find the state class (CounterModel), and invoke the method in the class.

```
ElevatedButton(
  onPressed: () {
    // calls the increment() method that triggers the action
    context.read<CounterModel>().increment();
  },
  child: Icon(Icons.add),
)
```

Use `context.read()` for actions that don't need to listen to changes
We used `setState(() => increment())` before.

Complete Example Structure

```
CounterModel (ChangeNotifier)
       ↓
ChangeNotifierProvider
       ↓
CounterScreen (Widget)
    ↓         ↓
Consumer   ElevatedButton
   ↓           ↓
 Text()   context.read()
```

1. User clicks the Button: The method in context.read<CounterModel>() is invoked.
2. In the CounterModel, the method invokes notifyListeners() method, and it triggers the action.
3. The builder in Consumer<CounterModel> is invoked.

Comparison with the setState() method

Provider	setState()
User clicks the Button: The context.read<CounterModel>().increment() is invoked.	User clicks the Button: The button calls setState(() => increment()).
In the CounterModel, the increment() method invokes notifyListeners() method, and it triggers the anction	The lambda expression increment() is called, and it triggers the action in the setState().
The builder in	The builder in

For the Provider approach, we should specify the ChangeNotifier Model (CounterModel) so that all the subclasses of `MyApp` can send notification using `context.read<CounterModel>` and receive the notification using `Consumer<CounterModel>(...)` to rebuild.

```
    ChangeNotifierProvider(
      create: (context) => CounterModel(),  // Create model
      child: MyApp(),
    ),
```

Consumer vs context.read()

Consumer	context.read()
Rebuilds when state changes	Does not rebuild
Used for displaying data	Used for triggering actions
Inside build() method	Inside event handlers

```
// receiver to redraw
Consumer<CounterModel>(
  builder: (context, model, child) => Text('${model.count}')
)

// sender to trigger notification
onPressed: () => context.read<CounterModel>().increment()

// ChangeNotifier Model
class CounterModel extends ChangeNotifier { ... }
```

Benefits of ChangeNotifier

Separation of Concerns

- Business logic separated from UI
- Easier to test and maintain

Reactive UI

- UI automatically updates when state changes
- No manual setState() calls needed

Shared State

- Multiple widgets can access the same state
- Foundation for complex architectures

Common Mistakes to Avoid

❌ Forgetting notifyListeners()

```
void increment() {
  _count++;
  // Missing: notifyListeners();
}
```

❌ Using Consumer for actions

```
// Don't do this:
Consumer<CounterModel>(
  builder: (context, model, child) => ElevatedButton(
    onPressed: () => model.increment(),  // ❌ Wrong!
    // context.read<CounterModel>().increment()
  )
)
```

Example (code/4. State Management/main.dart)

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

The MyApp widget is a MaterialApp that uses Provider using `ChangeNotifierProvider` class.

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'State Management with ChangeNotifier',
      home: ChangeNotifierProvider(
        // Create the CounterModel instance
        create: (context) => CounterModel(),
        child: const CounterScreen(),
      ),
    );
  }
}
```

The CounterScreen() widget has the Consumer<CounterModel> so that it can redraw from the notification with notifyListeners().

```dart
class CounterScreen extends StatelessWidget {
  const CounterScreen({super.key});

...

            // Consumer rebuilds only when the model changes
            Consumer<CounterModel>(
              builder: (context, counterModel, child) {
                return Text(
                  '${counterModel.count}',
                );
              },
```

The CounterScreen() widget has the context.read<CounterModel>() to call the methods to invoke notifyListeners() and trigger the redraw.

```dart
class CounterScreen extends StatelessWidget {
  const CounterScreen({super.key});

...

                // Reset button
                ElevatedButton(
                  onPressed: () {
                    // Access the model and call reset
                    context.read<CounterModel>().reset();
                  },
                  child: const Icon(Icons.refresh),
                ),
}
```