MVVM Pattern in Flutter

Model-View-ViewModel Software Architecture

Learning Objectives

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

- ✅ **Understand** what MVVM is and why it's used
- ✅ **Identify** the three components of MVVM
- ✅ **Implement** MVVM pattern in Flutter applications
- ✅ **Compare** MVVM with other architectural patterns
- ✅ **Apply** best practices for MVVM in real projects

What is MVVM?

Model-View-ViewModel is an architectural pattern that separates:

Real-world analogy:

Like a restaurant where:

- **Model** = Kitchen (data preparation)
- **View** = Dining room (customer experience)
- **ViewModel** = Waiter (communication bridge)

MVVM Components Overview

Component	Responsibility	Flutter Implementation
Model	Data structure	Dart classes
View	User Interface	Widgets
ViewModel	Business Logic	ChangeNotifier + State Management(Provider)

Why Use MVVM?

✅ Benefits:

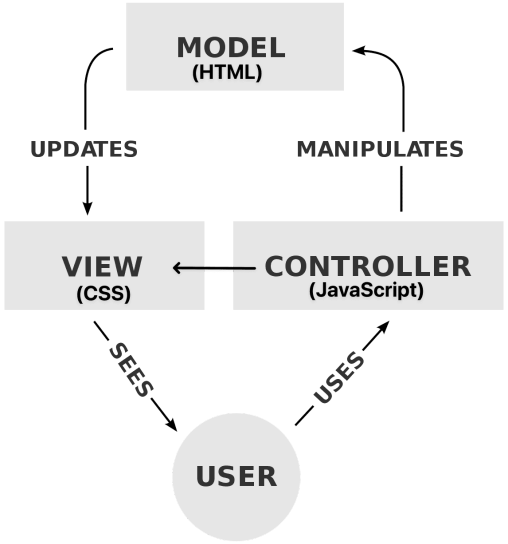
- **Separation of Concerns**
- **Testability** - Unit test business logic
- **Maintainability** - Easy to modify
- **Reusability** - ViewModels can be reused
- **Reactive UI** - Automatic updates

⚠️ When NOT to use:

- Very simple apps
- Static content only
- Learning/prototype apps
- When team lacks experience

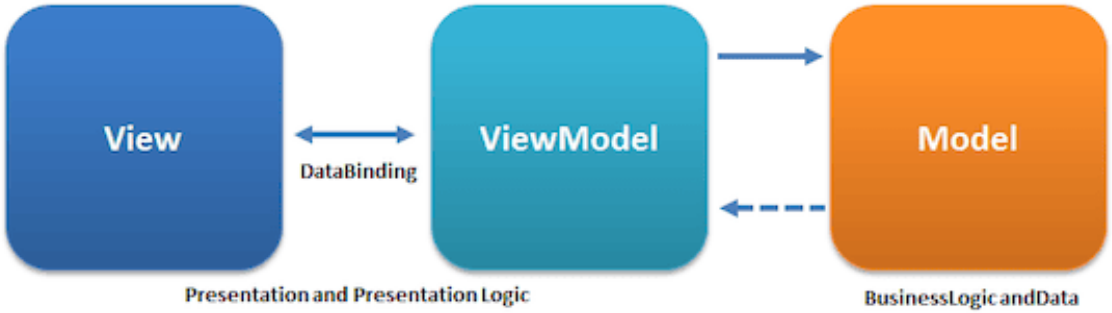
MVC vs MVVM

Pattern	View-Logic Coupling	Testability	Complexity	Flutter Usage
MVC	Medium	Medium	Low	★★
MVVM	Low	High	Medium	★★★★



- **MVC**
  - **Controller** handles input and updates both Model & View.
  - View often depends on Controller → **tighter coupling**.
  - Works fine for small apps, but scaling makes Controllers heavy.

- **MVVM**
  - **ViewModel** exposes data/state to the View.
  - View listens/reacts via **bindings** (observers, streams, notifies).
  - Decouples UI from logic → easier **testing** and **reuse**.

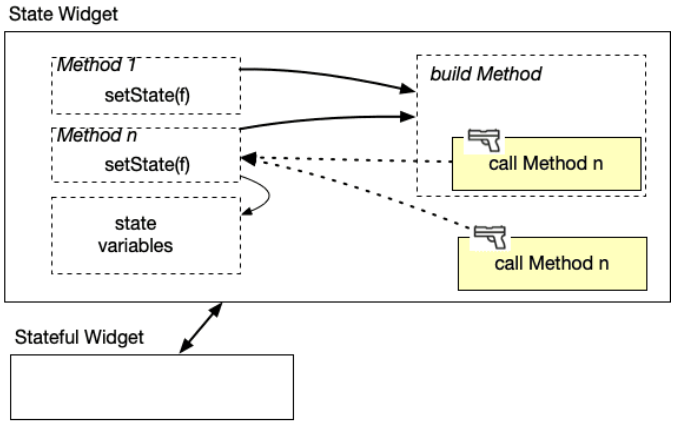


- Flutter apps handle diverse user inputs, such as gestures.
- **MVC**, designed originally for web applications, struggles with this level of complexity in Flutter.
- **MVVM** works better: it separates **Views** (widgets) and **Models** (data) using a **ViewModel** (state management / change notifier), making apps more maintainable.

SetState-Build Model vs MVVM

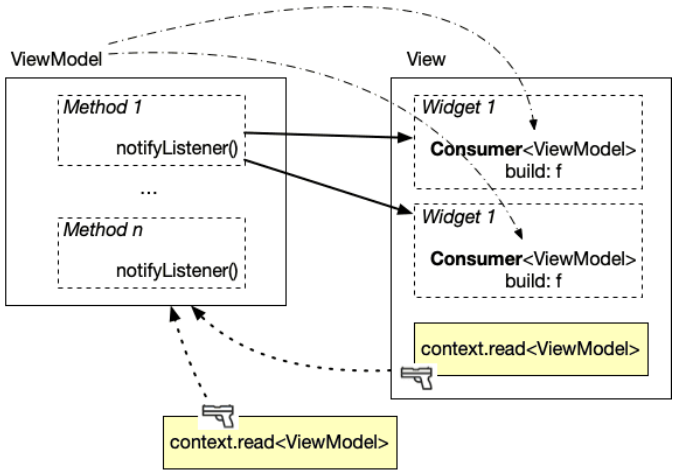
We have used setState-Build model to redraw Widgets.

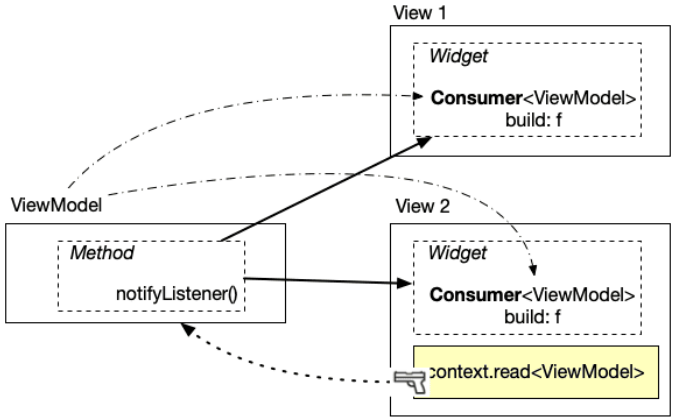
1. **Trigger** → Any widget method (or external caller) can invoke `setState()`
2. **Update** → Lambda updates state variables
3. **Redraw** → `build()` is called, UI rebuilds with new state



However, this model cannot trigger other widgets, so to redraw multiple widgets, we need to use the MVVM model.

1. **Trigger** → Widget calls `context.read<ViewModel>()` method
2. **Update** → ViewModel updates state & calls `notifyListeners()`
3. **Redraw** → `Consumer<ViewModel>` widgets rebuild on notification


ViewModel
View
Method 1
notifyListener()
...
Method n
notifyListener()
Widget 1
Consumer<ViewModel>
build: f
Widget 1
Consumer<ViewModel>
build: f
context.read<ViewModel>
context.read<ViewModel>


View 1
Widget
Consumer<ViewModel>
build: f
ViewModel
View 2
Method
notifyListener()
Widget
Consumer<ViewModel>
build: f
context.read<ViewModel>

Flutter MVVM Implementation Overview (Short Version)

1. MODEL Layer - Data Structure

```dart
// models/todo.dart
class Todo {
  final String id;
  final String title;
  final bool isCompleted;

  Todo({
    required this.id,
    required this.title,
    this.isCompleted = false,
  });
}
```

Model Characteristics:

- Pure Dart class (no Flutter dependencies)
- Only data and data-related methods

2. VIEWMODEL Layer - Business Logic

ViewModel Characteristics:

- Extends `ChangeNotifier` for state management
- Contains ALL business logic (also core data structrue List<Todo>)
- No UI dependencies (pure business code)
    - Separation of View through notifyListener().
- Uses `notifyListeners()` to update UI

ViewModel Business Methods

```dart
// viewmodels/todo_viewmodel.dart
class TodoViewModel extends ChangeNotifier {
  final List<Todo> _todos = [];

  // Read-only access to data
  List<Todo> get todos => List.unmodifiable(_todos);

  void addTodo(String title) {
    // Business Logic
    if (title.trim().isEmpty) return; // Validation
    final newTodo = Todo(...)
    _todos.add(newTodo);

    // notify to redarw
    notifyListeners(); // Notify UI to update
  }
```

3. VIEW Layer - User Interface

✅ View Characteristics:

- Pure UI code (widgets only)
- No business logic
- Delegates actions to ViewModel

Consumer - For Displaying Data

The Consumer<TodoViewModel> class has builder lambda exprssion to redraw itself.

```
// views/todo_view.dart
class TodoView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        children: [ _buildStatsSection() ... ],
      ),
    );
  }
// Use Consumer when UI needs to UPDATE automatically
Widget _buildStatsSection() {
  return Consumer<TodoViewModel>( // <-- Consumer<VM>
    builder: (context, viewModel, child) { ... }
  );
}
```

Trigger: context.read<ViewModel>()

```
// Use context.read() when you just need to TRIGGER actions
ElevatedButton(
  onPressed: () {
    // Trigger 1
    context.read<TodoViewModel>().addTodo(_controller.text);
    _controller.clear();
  },
  child: Text('Add Todo'),
),

IconButton(
  onPressed: () => // Trigger 2
  context.read<TodoViewModel>().removeTodo(todo.id),
  icon: Icon(Icons.delete),
),
```

✅ **No listener dependency → Better performance for actions**

The Magic: Consumer vs context.read()

Two ways to access ViewModel:

Purpose	Method	When to Use
Display changing data	`Consumer<T>`	Statistics, lists, counts
Trigger actions	`context.read<T>()`	Button clicks, form submits

App Setup - Connecting View & ViewModel

We use ChangeNotifierProvider to connect the ViewModel to the View.

```
// main.dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: ChangeNotifierProvider(
        create: (context) => TodoViewModel(), // Create ViewModel
        child: TodoView(),                    // Provide to View
      ),
    );
  }
}
```

MVVM Benefits

✅ **Separation of Concerns:**

- Model: Pure data structure
- ViewModel: Pure business logic
- View: Pure UI code

✅ **Maintainability:** Change business logic without touching UI
✅ **Reusability:** Same ViewModel for mobile & web views

✅ Testability:

```
void main() {
  test('should add todo', () {
    final viewModel = TodoViewModel();
    viewModel.addTodo('Test task');
    expect(viewModel.todos.length, 1);
    expect(viewModel.todos.first.title, 'Test task');
  });
}
```