Mobile Programming

Flutter UI Design & Patterns

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

MVP Pattern | Custom Widgets | Dynamic Layouts | Best Practices

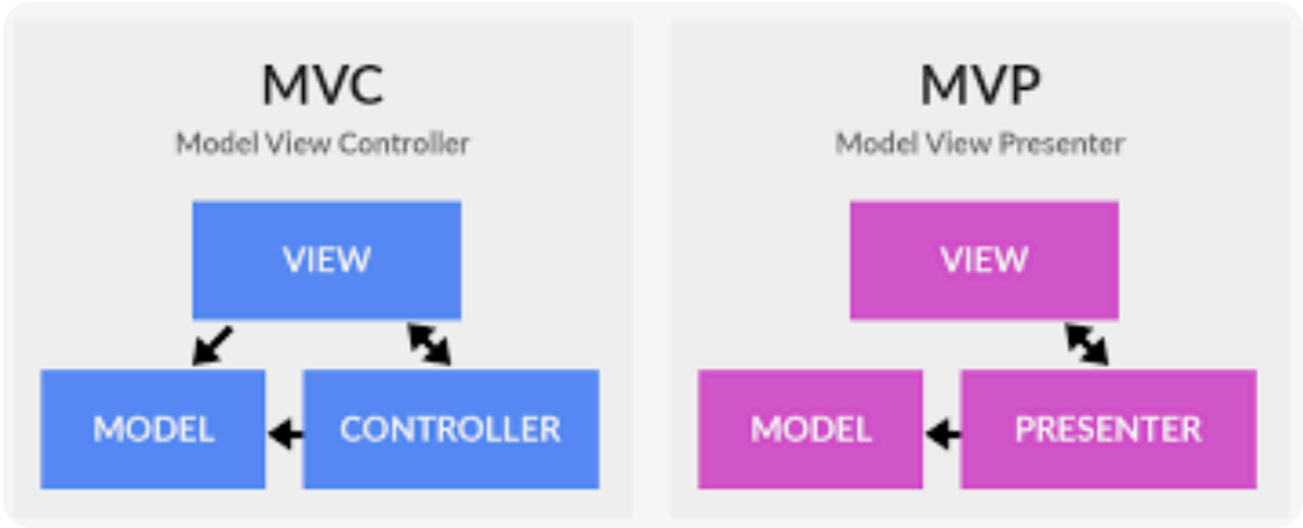


Table of Contents

1. Introduction to MVP Architecture in Flutter
2. Understanding MVP vs MVVM vs MVC
3. UI Components in Flutter
4. Custom UI: Extending Widgets
5. Building Custom Widgets: GroupView
6. Dynamic Layout Management
7. Best Practices for Flutter UI Design
8. Mini Lab: Building a Custom Dashboard
9. Summary & Key Takeaways

Introduction to MVP Architecture

MVP (Model-View-Presenter) is a derivative of MVC that focuses on **separating presentation logic** from the UI.



- **Model:** Data layer — business logic, API calls, database operations.
- **View:** Flutter widgets — passive, only displays what the Presenter tells it.
- **Presenter:** Mediator — receives user actions, updates Model, tells View what to display.

How MVP Works in Flutter

The Presenter is the **brain**; the View is the **face**.

View Responsibilities

- Display UI elements.
- Capture user interactions (taps, gestures).
- Call Presenter methods on user actions.
- Implement a contract interface (`IView`).

Presenter Responsibilities

- Process business logic.
- Fetch/update data from Model.
- Call `view.updateUI()` to push new state.
- Never directly builds widgets.

How MVP Works in Flutter

```dart
// View Contract
abstract class DashboardView {
  void showLoading();
  void showUser(User user);
  void showError(String msg);
}
// Presenter
class DashboardPresenter {
  final DashboardView view;
  final UserRepository repo;

  void loadUser(int id) async {
    view.showLoading();
    try {
      var user = await repo.getUser(id);
      view.showUser(user);
    } catch(e) {
      view.showError(e.toString());
    }
  }
}
```

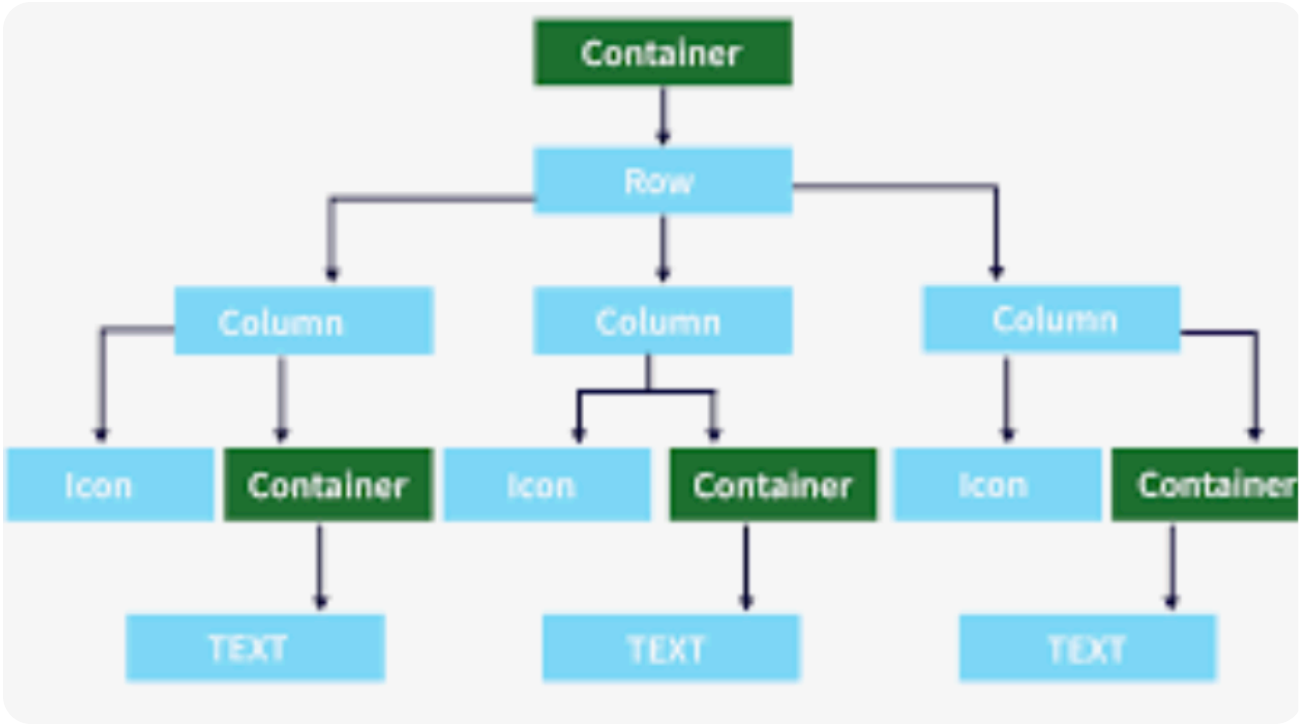
MVP vs MVVM vs MVC

Understanding the landscape of architectural patterns.

Aspect	MVC	MVP	MVVM
View	Knows about Model	Passive, has Presenter	Data binds to ViewModel
Communication	Direct View↔Model	View ↔ Presenter, Presenter ↔ Model	Data binding (Observable)
Testability	Moderate	High (Presenter is pure Dart)	High (ViewModel is testable)
Flutter Fit	Less common	Very common	Good with Provider/Riverpod

Flutter UI Component Hierarchy

Flutter is built on a **tree of widgets.** Understanding this hierarchy is fundamental.



Key Concept: Everything in Flutter is a widget — even `Scaffold`, `AppBar`, and `FloatingActionButton`.

Core UI Components in Flutter

Layout Widgets

- `Container` box model with padding, margin, decoration.
- `Row / Column` linear layouts (Flex).
- `Stack` layered positioning.
- `GridView / ListView` scrollable collections

Interactive Widgets

- `ElevatedButton` raised button with ink splash.
- `TextField` text input.
- `Slider / Switch` value pickers.
- `GestureDetector` raw touch handling.

Structural Widgets

- `Scaffold` — material page structure.
- `AppBar` — top navigation bar.
- `BottomNavigationBar` — tab switching.
- `Card` — elevated content container.

Content Widgets

- `Text` — rich text display.
- `Image` — asset/network image.
- `Icon` — material/design icons.
- `ProgressIndicator` — loading feedback.

Extending Widgets: Composition

```dart
// Wrap any widget with a loading overlay
class LoadingOverlay extends StatelessWidget {
  final bool isLoading;
  final Widget child;
  const LoadingOverlay({super.key, required this.isLoading, required this.child,});
  @override
  Widget build(BuildContext context) {
    return Stack(
      children: [
        child,
        if (isLoading)
          Container(
            color: Colors.black54,
            child: Center(
              child: CircularProgressIndicator(),
            ),
          ),
      ],
    );
  }
}
```

Building a Custom Widget: GroupView

A `GroupView` is a reusable container that groups related widgets with a title, optional subtitle, and consistent styling.

What it does:

- Groups related UI elements under a header.
- Provides consistent card styling.
- Supports optional trailing action (icon, button).
- Handles divider lines between items.

```dart
class GroupView extends StatelessWidget {
  final String title;
  final String? subtitle;
  final List<Widget> children;
  final Widget? trailing;

  const GroupView({
    super.key,required this.title,this.subtitle,required this.children,this.trailing,
  });
  @override
  Widget build(BuildContext context) {
    return Card(
      margin: EdgeInsets.symmetric(vertical: 8),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          ListTile(
            title: Text(title,
              style: TextStyle(fontWeight: FontWeight.bold)),
            subtitle: subtitle != null ? Text(subtitle!) : null,
            trailing: trailing,
          ),
          Divider(height: 1),
          ...children,
        ],
      ),
    );
  }
}
```

Using GroupView in Practice

```
GroupView(
  title: 'Account Settings', subtitle: 'Manage your profile',
  trailing: IconButton(
    icon: Icon(Icons.edit),
    onPressed: () {},
  ),
  children: [
    ListTile(
      leading: Icon(Icons.person),
      title: Text('Name'),
      subtitle: Text('Ali Najimi'),
      onTap: () {},
    ),
    ListTile(
      leading: Icon(Icons.email),
      title: Text('Email'),
      subtitle: Text('ali@example.com'),
      onTap: () {},
    ),
  ],
)
```

Dynamic Layout Management

Flutter's declarative nature makes dynamic layouts powerful and predictable.

Core Strategies:

1. **Conditional Widgets:** Show/hide widgets based on state.
2. **ListView.builder:** Render only visible items for performance.
3. **AnimatedSwitcher:** Smooth transitions between different content.
4. **LayoutBuilder:** Adapt layout based on available space.

```
@override
Widget build(BuildContext context) {
  return Column(
    children: [// Conditional widget rendering
      if (isLoading) CircularProgressIndicator(),
      if (error != null) Text('Error: $error'),
      if (data != null) DataListView(data: data),
    ],
  );
}
```

Dynamic Layout: ListView.builder

For large or unknown-length lists, `builder` is essential.

```
// Efficient rendering — only builds visible items
ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    final item = items[index];
    return ListTile(
      leading: CircleAvatar(child: Text('${index + 1}')),
      title: Text(item.title),
      subtitle: Text(item.description),
      onTap: () => onItemTap(item),
    );
  },
)
```

Performance: `ListView.builder` recycles widget instances as the user scrolls — critical for lists with 100+ items.

Dynamic Layout: LayoutBuilder & MediaQuery

Adapt your UI to screen size, orientation, and platform.

```
// Respond to parent size constraints
LayoutBuilder(
  builder: (context, constraints) {
    if (constraints.maxWidth > 600) {
      return _buildWideLayout();
    } else {
      return _buildNarrowLayout();
    }
  },
)
```

```
// Respond to screen size
MediaQuery.of(context).size;        // Width/Height
MediaQuery.of(context).orientation; // Portrait/Landscape
MediaQuery.of(context).devicePixelRatio;

// Convenience extension
final isTablet = MediaQuery.of(context).size.shortestSide >= 600;
final isLandscape = MediaQuery.of(context).orientation == Orientation.landscape;
```

Best Practices: Widget Design

1. **Keep widgets small and focused** — Single Responsibility Principle.
2. **Avoid deeply nested widgets** — Flatten your tree where possible.
3. **Use `const` constructors** — Reduces rebuild cost.
4. **Extract reusable UI patterns** — Create custom widget libraries.

Best Practices: Widget Design

Bad: Deeply nested, not reusable

```
@override
Widget build(BuildContext context) {
  return Container(
    decoration: BoxDecoration(
      color: Colors.white,
      borderRadius: BorderRadius.circular(12),
      boxShadow: [BoxShadow(...)],
    ),
    child: Column(
      children: [
        Padding(
          padding: EdgeInsets.all(16),
          child: Text('Title'),
        ),
        // ... many more levels
      ],
    ),
  );
}
```

Best Practices: Widget Design

Good: Extract to custom widget

```
class StyledCard extends StatelessWidget {
  final Widget child;
  const StyledCard({super.key, required this.child});

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(
        color: Colors.white,
        borderRadius: BorderRadius.circular(12),
        boxShadow: [BoxShadow(...)],
      ),
      child: child,
    );
  }
}
```

Best Practices: State Management for UI

Key rule: Separate UI state from business data.

Do:

- Keep state close to where it's used.
- Use `StatefulWidget` for local UI state (animations, toggles).
- Lift state up only when needed by multiple widgets.

Don't:

- Put all state in the top-level widget.
- Store UI state in `StatelessWidget`, it can't rebuild.
- Use `setState()` for complex/async state.

```dart
// Lift state only when necessary
class ParentWidget extends StatelessWidget {
  final String data; // Shared state

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        ChildA(data: data), // Passed down
        ChildB(data: data), // Passed down
        ChildC(data: data), // Passed down
      ],
    );
  }
}

// Use setState for simple local state only
class ToggleWidget extends StatefulWidget {
  const ToggleWidget({super.key});

  @override
  State<ToggleWidget> createState() => _ToggleWidgetState();
}

class _ToggleWidgetState extends State<ToggleWidget> {
  bool _isExpanded = false; // Local UI state
  // ...
}
```

Best Practices: Performance Optimization

1. `const` **widgets** — Mark widgets that don't change as `const`.
2. **Avoid** `setState()` **in** `build()` — Only call it in response to events.
3. **Use** `RepaintBoundary` — Isolate expensive custom painters.
4. **Avoid anonymous function listeners** — Store references for disposal.

Best Practices: Performance Optimization

Good: const prevents unnecessary rebuilds

```
Widget build(BuildContext context) {
  return Column(
    children: [
      const Text('Static label'), // Never rebuilds
      Text(dynamicValue),         // Only this rebuilds
    ],
  );
}
```

Best Practices: Performance Optimization

Bad: Rebuilds on every state change

```
Widget build(BuildContext context) {
  return Text('Hello ${widget.name}');
}
```

Best Practices: Performance Optimization

Good: Use RepaintBoundary for expensive custom paint

```
RepaintBoundary(
  child: CustomPaint(
    painter: MyExpensivePainter(),
  ),
)
```

Mini Lab

Building a Custom Dashboard Widget

Objective: Build a reusable dashboard card system using MVP + custom widgets.

Steps:

1. Define `DashboardCard` widget with title, icon, value, and trend.
2. Create `DashboardView` interface with `showCards(List<CardModel>)`.
3. Implement `DashboardPresenter` with mock data loading.
4. Use `GridView.builder` to render cards dynamically.
5. Add `RefreshIndicator` for pull-to-refresh.
6. Use mock data to render dashboard.

Mini Lab: Step-by-Step Code

Step 1: Model

```
class CardModel {
  String title;
  IconData icon;
  String value;
  double trend;
}
```

Mini Lab: Step-by-Step Code

Step 2: View Contract

```
abstract class DashboardViewContract {
  void showLoading();
  void showCards(List<CardModel> cards);
  void showError(String msg);
}
```

Mini Lab: Step-by-Step Code

Step 3: Presenter

```
class DashboardPresenter {
  final DashboardViewContract view;
  DashboardPresenter(this.view);

  void loadCards() async {
    view.showLoading();
    await Future.delayed(Duration(seconds: 1)); // Simulate API
    view.showCards([
      CardModel(title: 'Revenue', icon: Icons.attach_money, value: '\$12,450', trend: 0.12),
      CardModel(title: 'Users', icon: Icons.people, value: '1,234', trend: 0.05),
    ]);
  }
}
```

Mini Lab: Step-by-Step Code

Step 4: Widget

```dart
class DashboardScreen extends StatefulWidget {
  const DashboardScreen({super.key});

  @override
  State<DashboardScreen> createState() => _DashboardScreenState();
}
```

Mini Lab: Step-by-Step Code

Step 4: Widget

```
class _DashboardScreenState extends State<DashboardScreen> implements DashboardViewContract {
  late final DashboardPresenter _presenter;
  List<CardModel> _cards = [];
  bool _loading = true;

  @override
  void initState() {
    super.initState();
    _presenter = DashboardPresenter(this);
    _presenter.loadCards();
  }

  @override
  void showLoading() => setState(() => _loading = true);
  @override
  void showCards(List<CardModel> cards) => setState(() { _loading = false; _cards = cards; });
  @override
  void showError(String msg) => ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text(msg)));
```

Mini Lab: Step-by-Step Code

Step 4: Widget (Cont.)

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Dashboard')),
      body: _loading ? Center(child: CircularProgressIndicator())
          : GridView.builder(
              padding: EdgeInsets.all(16),
              gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
                crossAxisCount: MediaQuery.of(context).size.width > 600 ? 3 : 2,
                crossAxisSpacing: 12, mainAxisSpacing: 12,
              ),
              itemCount: _cards.length,
              itemBuilder: (ctx, i) => DashboardCard(model: _cards[i]),
            ),
    );
  }
}
```

Summary: Key Takeaways

Architecture

- MVP separates **View** (widgets) from **Presenter** (logic) and **Model** (data).
- Use contracts/interfaces for testability.

Custom Widgets

- Prefer **composition** over modification.
- Extract reusable patterns (`GroupView`, `StyledCard`).

Dynamic Layouts

- Use `ListView.builder` for large lists.
- Use `LayoutBuilder` / `MediaQuery` for responsive design.

Performance

- Use `const` constructors where possible.
- Avoid rebuilding widgets unnecessarily.
- Use `RepaintBoundary` for expensive custom paint.

Best Practices

- Keep widgets **small and focused**.
- Lift state only when **necessary**.
- Separate **UI state** from **business data**.

Thank You! 🙌

Questions?

"A Flutter developer walks into a bar.
The bartender says, 'What would you like?'
The developer replies, 'I'll take a Column, place a Text in it,
wrap it in a Card, and call it a day.'"