

# Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 3  
Architecture Pattern and Clean Code

**Lecture 7**



Chapter  
**07**

## Table Of Content

- Introduction
- Architecture Pattern
- Types of Architectures
- Tips and Tricks for Clean Code



# Introduction

# Introduction

## What is Architecture Pattern?

### Definition:

- A design pattern that helps organize code into separate concerns
- Defines how code is structured across layers (UI, logic, data).
- Examples: MVC, MVP, MVVM, Clean Architecture, Hexagonal Architecture.
- Helps manage app complexity, testability, and scalability.



# Introduction

## Why Care About Architecture?

- Helps organize your codebase as your app grows.
- Easier to maintain, debug, and extend features.
- Promotes scalability and reusability.
- Makes it easier for teams to collaborate.
- Enables writing unit and integration tests.



# Introduction

## What is Software Architecture?

- The structure of your code: how responsibilities are divided.
- Just like a building's blueprint – architecture guides development.
- Separates concerns: UI, logic, and data shouldn't mix.
- Helps enforce rules and patterns in how code is written.



# **Architecture Pattern**

# Architecture Pattern

## Key Terms to Know

- **UI (User Interface):** Widgets users interact with.
- **Business Logic:** What your app does (rules, decisions).
- **Data Layer:** Communicates with APIs or databases.
- **State Management:** How UI reacts to data changes.



# Architecture Pattern

## Key Terms to Know

- **UI (User Interface):** Widgets users interact with.
- **Business Logic:** What your app does (rules, decisions).
- **Data Layer:** Communicates with APIs or databases.
- **State Management:** How UI reacts to data changes.



# Architecture Pattern

## Anatomy of a Flutter App

- **Widgets:** Building Widgets of UI.
- **Models:** Represent data structures (e.g., **User**, **Product**).
- **Services:** API or DB communication logic.
- **Controllers/ViewModels/Blocs:** Handle logic.
- **Repositories:** Bridge between data sources and logic.



# **Types of Architecture Pattern**

# Architecture Pattern

## What is MVC (Model-View-Controller)?

- **Model:** Manages data and rules.
- **View:** The UI layer (Flutter widgets).
- **Controller:** Connects Model and View, updates data.
- Example: User taps button -> Controller updates Model -> View refreshes.



```
class User {  
    String name;  
    User(this.name);  
}  
  
class UserController {  
    User user = User("Youssef");  
    void changeName(String name) {  
        user.name = name;  
    }  
}  
  
class UserView extends StatelessWidget {  
    final UserController controller;  
    UserView(this.controller);  
  
    @override  
    Widget build(BuildContext context) {  
        return Text(controller.user.name);  
    }  
}
```

# Architecture Pattern

## MVC - Pros & Cons

### ✓ Pros:

- Simple for small apps.
- Easy to understand.

### ✗ Cons:

- Controller can become bloated.
- Difficult to scale.
- Not test-friendly.



# Architecture Pattern

## What is MVP (Model-View-Presenter)?

- **Model:** Handles data.
- **View:** Stateless UI.
- **Presenter:** Handles logic and updates the view indirectly.
- Decouples View from logic.
- Useful for apps needing high testability.



```
abstract class UserView {  
    void updateUserName(String name);  
}
```

```
class UserPresenter {  
    final UserView view;  
    UserPresenter(this.view);  
  
    void loadUser() {  
        view.updateUserName("Ahmed");  
    }  
}
```

```
class UserWidget extends StatelessWidget implements UserView {  
    String name = "";  
    late final presenter = UserPresenter(this);  
  
    @override  
    Widget build(BuildContext context) {  
        presenter.loadUser();  
        return Text(name);  
    }  
  
    @override  
    void updateUserName(String name) {  
        this.name = name;  
    }  
}
```

# Architecture Pattern

## MVP - Pros & Cons

### ✓ Pros:

- More testable than MVC.
- View logic is separate from UI.

### ✗ Cons:

- Slightly more boilerplate.
- View can become complex.



# Architecture Pattern

## What is MVVM (Model-View-ViewModel)?

- **Model:** Handles the data.
- **View:** Flutter UI.
- **ViewModel:** Handles logic, exposes observable state to View.
- View reacts to changes using Provider or ValueNotifier.



```
class UserViewModel extends ChangeNotifier {
    String _name = "Ali";
    String get name => _name;

    void updateName(String newName) {
        _name = newName;
        notifyListeners();
    }
}

class UserScreen extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return ChangeNotifierProvider(
            create: (_) => UserViewModel(),
            child: Consumer<UserViewModel>(
                builder: (_, viewModel, __) => Column(
                    children: [
                        Text(viewModel.name),
                        ElevatedButton(
                            onPressed: () => viewModel.updateName("Omar"),
                            child: Text("Change Name"),
                        ),
                    ],
                ),
            ),
        );
    }
}
```

# Architecture Pattern

## MVVM - Pros & Cons

### ✓ Pros:

- Easy to test ViewModel.
- View is reactive.
- Good for medium-large apps.

### ✗ Cons:

- Extra boilerplate.
- Needs understanding of state management.

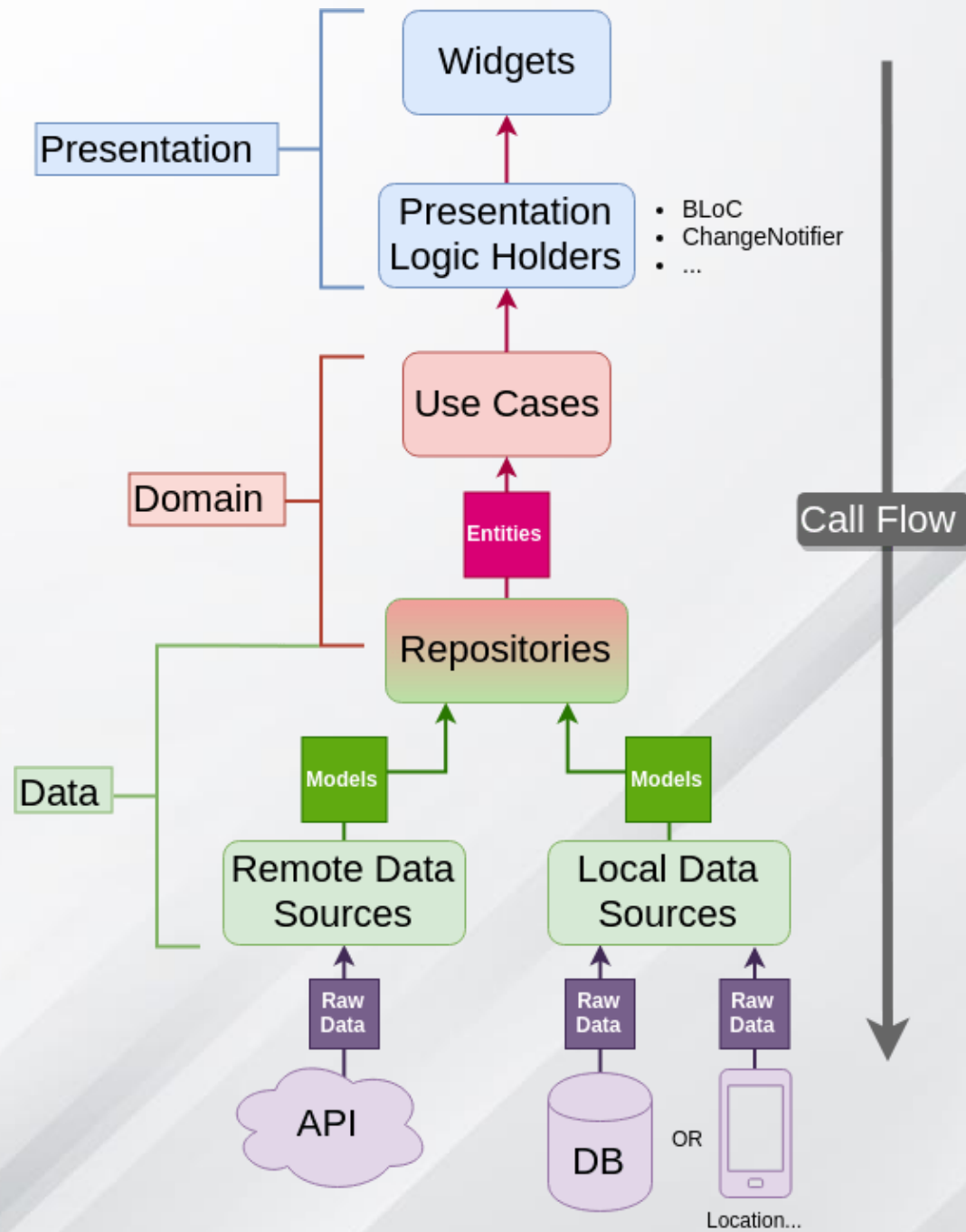


# Architecture Pattern

## What is Clean Architecture?

- Introduced by Uncle Bob (Robert C. Martin).
- Separates app into layers:
  - **Entities:** Core business rules.
  - **UseCases:** Application-specific rules.
  - **Repositories:** Data access.
  - **Data Layer:** API, DB, services.
- Direction of flow: UI -> UseCase -> Repository -> Data





# Architecture Pattern

## Clean Architecture Layers Explained

- **Domain:**
  - Entities (data classes)
  - UseCases (business rules)
- **Data:**
  - Remote/API or local/DB
  - Mappers & DTOs
- **Presentation:**
  - UI Widgets
  - Bloc/ViewModel + State



# Architecture Pattern

## Clean Architecture Folder Structure

```
lib/  
  core/  
  features/  
    login/  
      domain/  
        entities/  
        usecases/  
        repositories/  
      data/  
        models/  
        datasources/  
        repositories_impl/  
      presentation/  
        bloc/  
        pages/
```



```
final loginProvider = Provider((ref) => LoginUseCase(ref.read(authRepoProvider)));
```

```
class LoginUseCase {  
  final AuthRepository repo;  
  LoginUseCase(this.repo);  
  
  Future<User> execute(String email, String pwd) {  
    return repo.login(email, pwd);  
  }  
}
```

```
final loginNotifierProvider = StateNotifierProvider<LoginNotifier, AsyncValue<User>>((ref) {  
    final loginUseCase = ref.read(loginProvider);  
    return LoginNotifier(loginUseCase);  
});
```

```
class LoginNotifier extends StateNotifier<AsyncValue<User>> {  
    final LoginUseCase useCase;  
    LoginNotifier(this.useCase) : super(const AsyncValue.data(User.empty()));
```

```
    Future<void> login(String email, String pwd) async {  
        state = const AsyncValue.loading();  
        try {  
            final user = await useCase.execute(email, pwd);  
            state = AsyncValue.data(user);  
        } catch (e, st) {  
            state = AsyncValue.error(e, st);  
        }  
    }  
}
```

# Architecture Pattern

## Clean Architecture - Pros & Cons

### ✓ Pros:

- Highly testable and scalable.
- Strong separation of concerns.
- Encourages best practices.

### ✗ Cons:

- Steep learning curve.
- More boilerplate and structure.
- May feel overkill for small apps.



# Architecture Pattern

## Why State Management Matters

- Flutter UI is reactive; it rebuilds on state change.
- Bad state management = spaghetti code.
- Architecture patterns require proper state handling to work well.



# Architecture Pattern

## Common State Management Options

| Option     | Pattern Fit      | Easy? | Powerful? |
|------------|------------------|-------|-----------|
| setState   | MVC              | ✓     | ✗         |
| Provider   | MVVM, Clean Arch | ✓     | ✓         |
| Riverpod   | Clean Arch       | ✓ ✓   | ✓ ✓       |
| Bloc/Cubit | MVP, Clean Arch  | ✗     | ✓ ✓       |
| GetX       | Custom           | ✓ ✓   | ✗         |



```
final counterProvider = ChangeNotifierProvider((_) => CounterViewModel());
```

```
class CounterViewModel extends ChangeNotifier {  
  int count = 0;  
  void increment() {  
    count++;  
    notifyListeners();  
  }  
}
```

```
class CounterView extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    final vm = context.watch<CounterViewModel>();  
    return Column(  
      children: [  
        Text('${vm.count}'),  
        ElevatedButton(onPressed: vm.increment, child: Text('Add')),  
      ],  
    );  
  }  
}
```

# Architecture Pattern

## Folder Structure Best Practices

- Keep reusable parts in **shared**
- Each feature is isolated
- Clean code = faster onboarding

```
lib/  
  core/  
  features/  
    authentication/  
      domain/  
      data/  
      presentation/  
  shared/  
    widgets/  
    themes/  
    utils/
```



# Architecture Pattern

## Common Mistakes in Flutter Architecture

- Mixing business logic with UI (anti-pattern)
- Using setState everywhere
- No separation between layers
- Ignoring testability
- Premature over-engineering



# Architecture Pattern

## Real-World Tools for Flutter Architecture

- **get\_it:** Dependency injection
- **freezed:** Immutable classes and union types
- **json\_serializable:** Code generation for models
- **riverpod:** State management
- **mason:** Scaffolding project templates



# Architecture Pattern

## When to Use What?

- Start small → MVC or MVVM with Provider
- As app grows → Clean Architecture with Bloc/Riverpod
- Team of 3+ developers? Use Clean Arch or MVVM



# Architecture Pattern

## Summary of Patterns

| Pattern    | Use Case                 | Pros               | Cons                |
|------------|--------------------------|--------------------|---------------------|
| MVC        | Small apps               | Simple             | Hard to scale       |
| MVP        | Moderate UI complexity   | Testable           | View can get messy  |
| MVVM       | Reactive UI              | Easy with Provider | Needs understanding |
| Clean Arch | Large apps               | Very scalable      | Boilerplate         |
| Hex Arch   | Plug-and-play interfaces | Very flexible      | Advanced            |



# **Tips And Tricks for Clean Code**

# Clean Code

## Clean Code Principles

Writing clean code helps others (and your future self) understand and maintain your work easily.

### Core ideas:

- Clear naming
- Small functions
- Single responsibility
- Avoid duplication
- Code should read like prose



# Clean Code

## Tip #1 - Use Meaningful Names

- Bad: `var x = getData();`
- Good: `var userProfile = fetchUserProfile();`

```
// ❌ Not clear  
final c = TextEditingController();  
  
// ✅ Better naming  
final emailController = TextEditingController();
```

Use names that reflect the purpose, not the type.

- ✅ Avoid abbreviations unless they're widely accepted.



# Clean Code

## Keep Functions Short

- Ideally 10–20 lines max.
- One function = one purpose.

✓ Good Flutter Example:

```
void submitForm() {  
  if (!_formKey.currentState!.validate()) return;  
  _saveData();  
}  
  
void _saveData() {  
  final user = buildUser();  
  userRepository.save(user);  
}
```

✗ Bad:

```
void submitForm() {  
  if (_formKey.currentState!.validate()) {  
    final user = buildUser();  
    userRepository.save(user);  
    Navigator.push(...);  
    showSnackBar();  
  }  
}
```

# Clean Code

## Tip #3 - Use Constants, Not Magic Numbers

- Makes layout more readable and consistent across the app

✗ Bad:

```
Padding(  
    padding: EdgeInsets.all(16),  
)
```

✓ Good:

```
const defaultPadding = 16.0;  
  
Padding(  
    padding: EdgeInsets.all(defaultPadding),  
)
```

# Clean Code

## Tip #4 - Avoid Deep Nesting

✖ Bad:

```
if (isLoggedIn) {  
  if (user.isVerified) {  
    if (!user.isBanned) {  
      navigateToDashboard();  
    }  
  }  
}
```

✔ Good:

```
if (!isLoggedIn || !user.isVerified || user.isBanned) return  
navigateToDashboard();
```

# Clean Code

## Tip #5 - Comment Why, Not What

✗ Bad:

```
// set state  
setState() {  
  isLoading = true;  
});
```

✓ Good:

```
// Show loading spinner while data fetch is in progress  
setState() {  
  isLoading = true;  
});
```

# Clean Code

## Clean Code - Avoid Over-Commenting

- Comments should explain **why**, not repeat **what**.

✗ Redundant:

```
// Set value to true  
isLoading = true;
```

✓ Better:

```
// Show loading indicator while fetching data  
isLoading = true;
```

✓ Even better: code that doesn't need a comment:

```
startLoading();
```

# Clean Code

## Clean Code - DRY Principle

DRY = Don't Repeat Yourself

- Avoid duplicating logic or widgets.
- Extract shared logic into functions or classes.

✖ Bad:

```
Text("Welcome"),  
SizedBox(height: 8),  
Text("Welcome"),
```

✔ Good:

```
Widget welcomeText() => Text("Welcome");  
  
Column(  
  children: [  
    welcomeText(),  
    SizedBox(height: 8),  
    welcomeText(),  
  ],  
);
```

# Clean Code

## Clean Code - Widget Extraction

- Keep UI readable by splitting widgets into components.

✖ Bad:

```
return Column(
  children: [
    Text("Name"),
    TextField(...),
    Text("Email"),
    TextField(...),
  ],
);
```

✔ Good:

```
return Column(
  children: [
    LabeledInput(label: "Name"),
    LabeledInput(label: "Email"),
  ],
);

class LabeledInput extends StatelessWidget {
  final String label;
  const LabeledInput({required this.label});

  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text(label),
        TextField(),
      ],
    );
  }
}
```

# Clean Code

## Clean Code - Extension Methods

- Use extensions to keep utility logic clean and reusable.

```
extension StringValidation on String {  
    bool get isValidEmail => contains('@') && contains(".");  
}  
  
final email = "user@test.com";  
print(email.isValidEmail); // true
```

# **What is Next ?**



**Thank You**