Chapter 10:
Advanced Flutter development concepts

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Problem & Context:** the challenge of **scaling** Flutter apps.

As a Flutter application grows, it must remain:

Scalable: support increasing features, widgets, and codebase complexity.

Maintainable: keep the code easy to understand, modify, and evolve over time.

Well-structured: organize the project with clear responsibilities and avoid tight coupling.

Collaborative: enable multiple developers to work efficiently with fewer conflicts.

Reliable: maintain quality, performance, and testability as the application evolves.

Software architecture in Flutter

- **Problem & Context:** the challenge of **scaling** Flutter apps.

How to design a Flutter application that can scale without becoming **complex**, **fragile**, or **difficult to maintain and test** ?

A **solid software architecture** for scalable, clean, and maintainable applications.

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Design principles for a scalable app :**

  - **Separation of concerns:** separate UI, business logic, and data management responsibilities.

  - **Modularity & reusability:** build independent and reusable components.

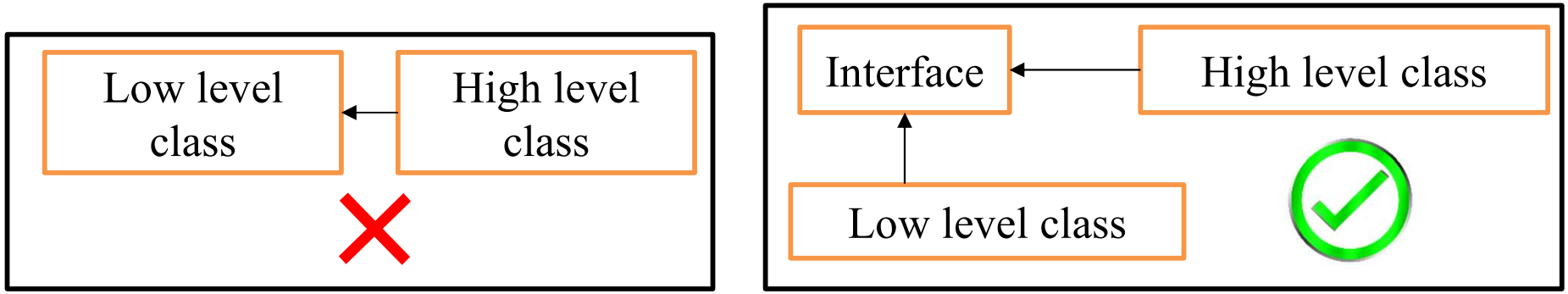
  - **High cohesion & low coupling:** keep related functionalities together while minimizing dependencies.

  - **Single source of truth:** maintain a unique and consistent source for application data and state.

Software architecture in Flutter

- **Design principles for a scalable app :**

➢ **Dependency inversion:** depend on abstractions rather than concrete implementations.



➢ **Unidirectional data flow:** ensure a predictable flow of data from state changes to UI updates.

➢ **Testability:** design components that can be verified independently.

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Common architectural patterns:** provide practical ways to apply software design principles.

  - **MVC**: Model / View / Controller.
  - **MVVM**: Model / View / ViewModel.
  - **Clean Architecture**.

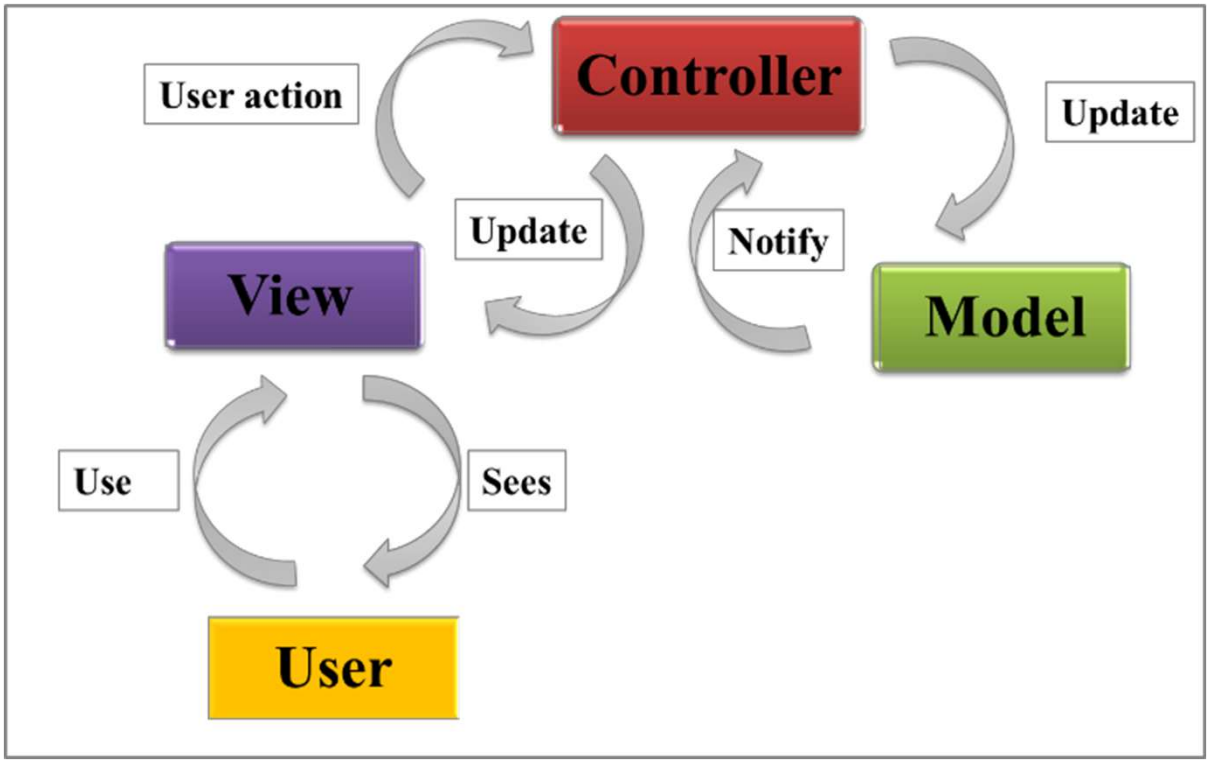
Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Common architectural patterns: MVC.**

  One of the earliest patterns, designed to separate an application into three components with distinct responsibilities.



- Separate UI, logic, and data management to improve code organization and maintainability.

Software architecture in Flutter

- **Common architectural patterns: MVC.**

Example: Counter app.

counter_model.dart

```
class  CounterModel {
          int _counter = 0;

          int get counter => _counter;
          void incrementCounter() {_counter++;}
          void decrementCounter() {_counter--;}}
```

counter_controller.dart

```
class CounterController {
final Model _model = Model();
int get counter => _model.counter;
void incrementCounter() {_model.incrementCounter();}
void decrementCounter() {_model.decrementCounter();}}
```

Software architecture in Flutter

- **Common architectural patterns: MVC.**

  Example: Counter app.

counter_view.dart OR counter_screen.dart

```
IconButton(icon:
                Icon(Icons.remove),
                onPressed:(){_controller.decrementCounter();}
            );
Text('${_controller.counter}'),
```

Software architecture in Flutter

- **Common architectural patterns: MVC.**

View: responsible for the user interface and presentation. Displays data to the user & captures user interactions.

Controller: acts as the mediator between *View* and *Model*. Handles user actions, coordinates application logic & updates the *View* based on data changes.

Model: represents the application data and business logic. Manages data sources & contains domain-related rules.

Software architecture in Flutter

- **Common architectural patterns: MVC.**

➢ **Advantages:**

- Simple and easy to understand.
- Clear separation between UI and data.
- Suitable for small to medium-sized applications.

➢ **Limitations:**

- *Controller* can become too large as the application Grows.
- Business logic may become difficult to isolate.
- Less suitable for large-scale applications.

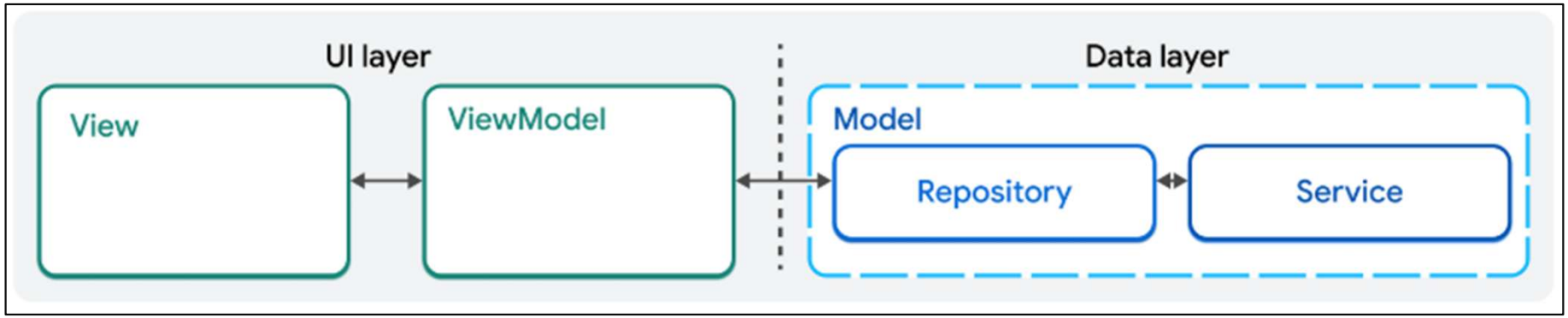
Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC**
     - **MVVM**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Common architectural patterns: MVVM.**
- Separates UI, state management, and data handling

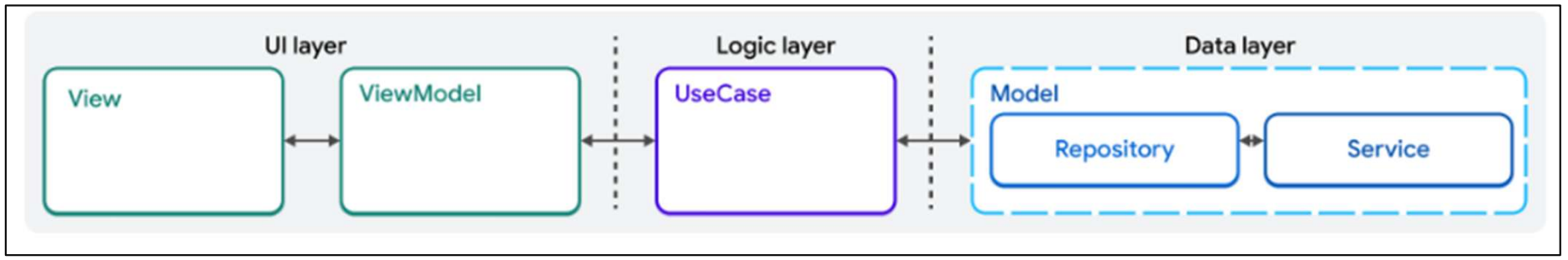
  - **View:** represents the UI + Displays state and forwards user actions.
  - **ViewModel:** contains UI-related logic +Transforms data into UI state + Handles user actions and updates the View.
  - **Model:** represents application data + Handles data access through repositories/services.



Software architecture in Flutter

- **Common architectural patterns: MVVM.**
- Extending MVVM for large-scale applications =>

  Introducing the Domain layer.



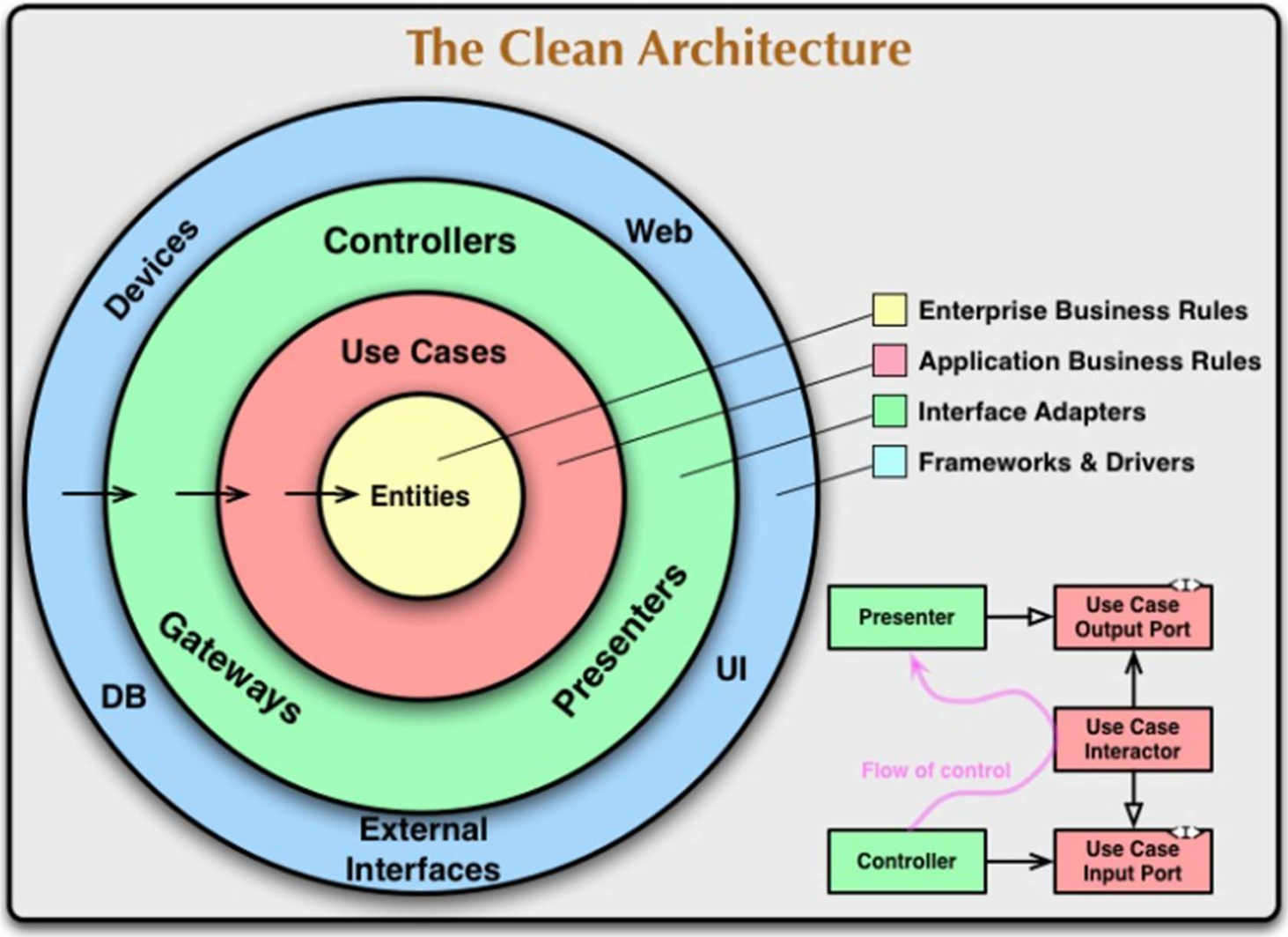
- ViewModels focus **only** on UI state management.
- Use Cases encapsulate application rules and workflows.

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC.**
     - **MVVM.**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

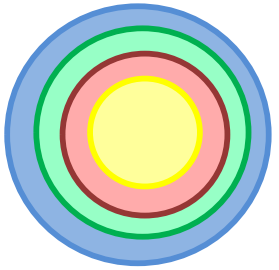
- **Common architectural patterns: Clean Architecture**



The Clean Architecture according to Uncle Bob

Software architecture in Flutter

- **Common architectural patterns: Clean Architecture**
  - **Entities:** core business objects and enterprise rules. ***The most stable layer.*** Exp: user, task, …
  - **Use Cases:** application-specific business logic and workflows. Exp: LoginUser, AddTask, …
  - **Interface Adapters:** convert data between the use cases and external layers (UI, controllers, presenters, repositories). Exp: BLoC, ViewModel, …
  - **Frameworks & Drivers:** external technologies and implementation details. ***The least stable layer.*** Exp: Flutter, Firebase, REST APIs, databases,…

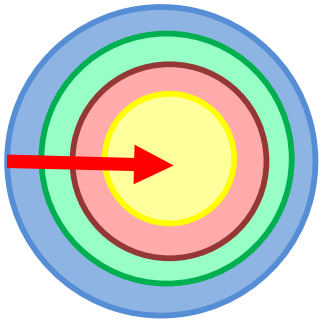


The Clean Architecture according to Uncle Bob

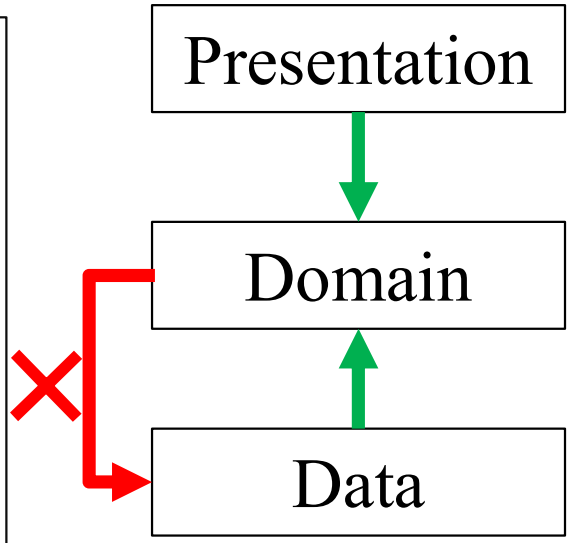
Software architecture in Flutter

- **Common architectural patterns: Clean Architecture**

Dependency rule: the flow of code dependencies must always point inward.

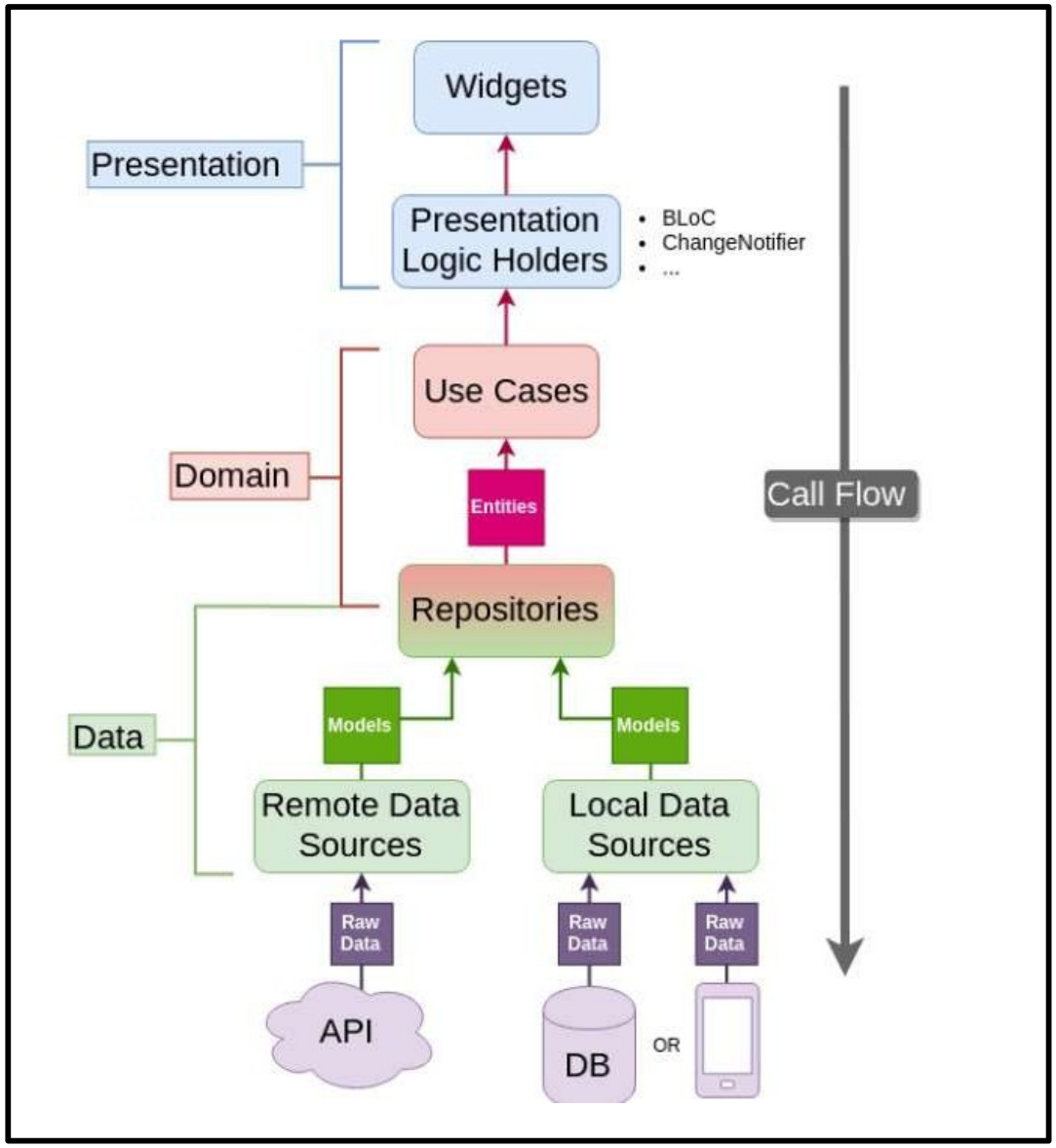


- Inner layers do not depend on outer layers.
- Business logic (domain) remains independent of UI, databases, and frameworks.
- External technologies can be replaced without affecting the core application.



Software architecture in Flutter

- **Common architectural patterns: Clean Architecture**



Layered representation of Clean Architecture for Flutter app.

⚠ ***Data flow*** ***≠*** ***Dependency rule***

Software architecture in Flutter

- **Common architectural patterns: impact of changing data sources** (replacing Firebase with a REST API).

- **Traditional MVVM:**
  - ✓ Changes often affect multiple layers: Repository, ViewModel, ….

- **Clean Architecture:**
  - ✓ Changes are isolated to the Data layer only.
  - ✓ The Domain & Presentation layer remain unchanged.

Software architecture in Flutter

- **Common architectural patterns:**

MVVM vs Clean architecture

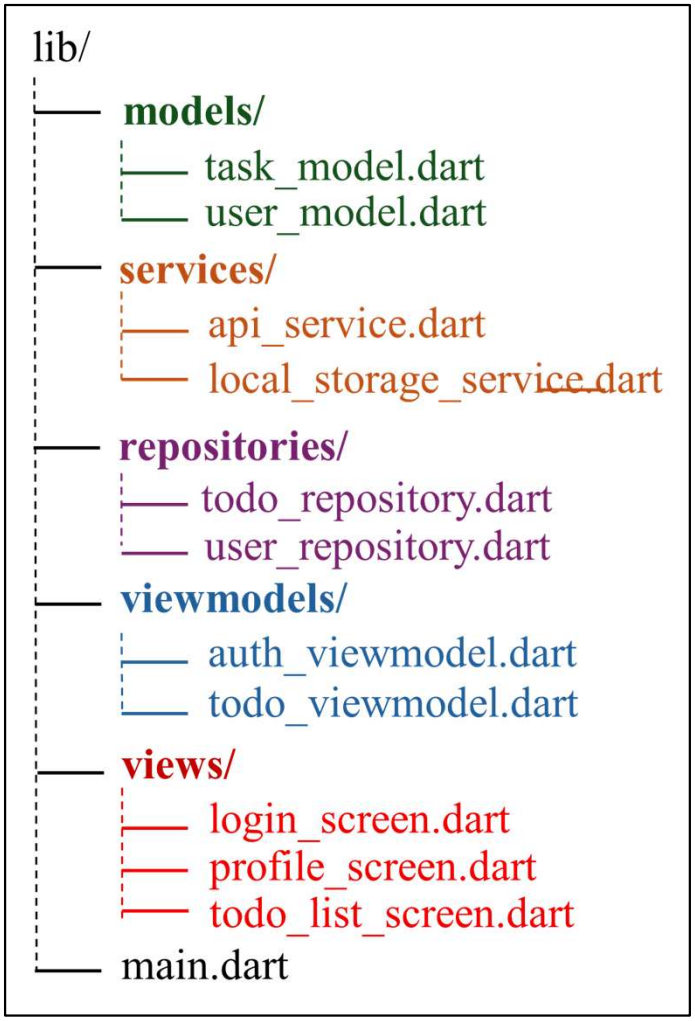
MVVM	Clean Architecture
➢ Organizes responsibilities. ➢ Defines **who does what**.	➢ Organizes responsibilities. & controls dependencies. ➢ Defines **who does what** and **who is allowed to know whom**.

Plan

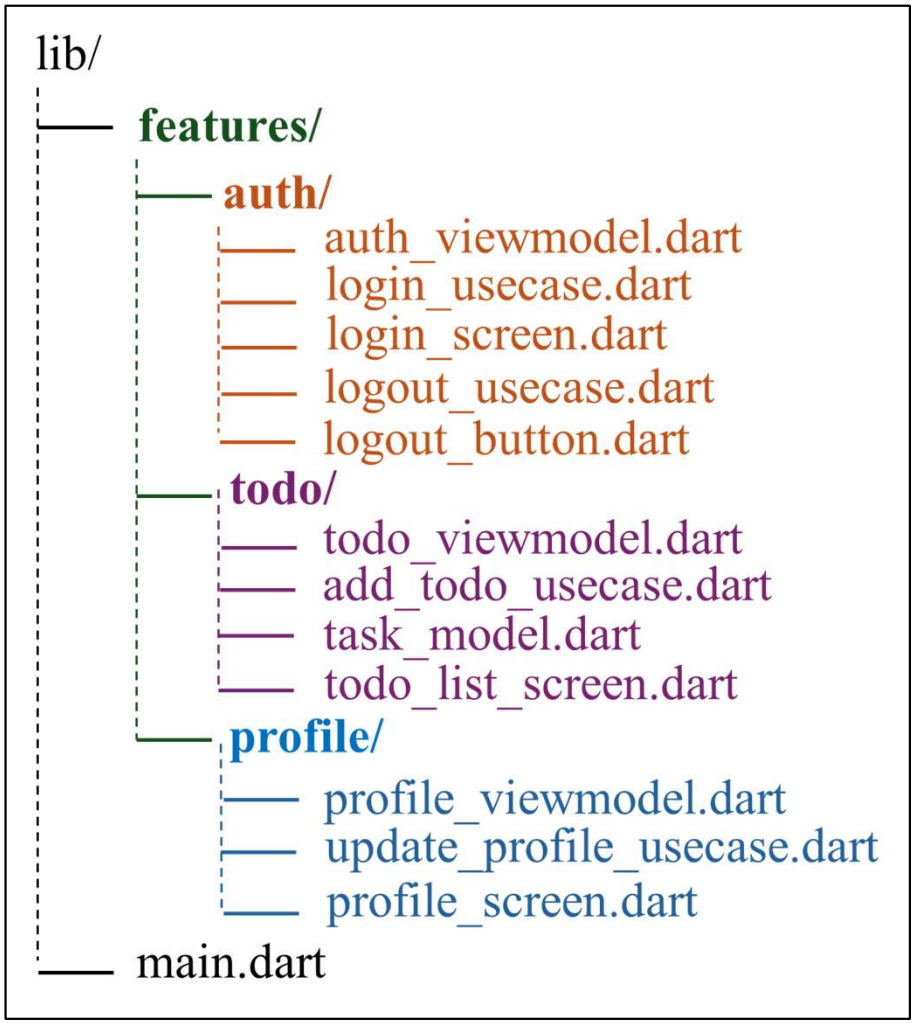
1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC.**
     - **MVVM.**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Software architecture in Flutter

- **Common architectural patterns: project structure.**



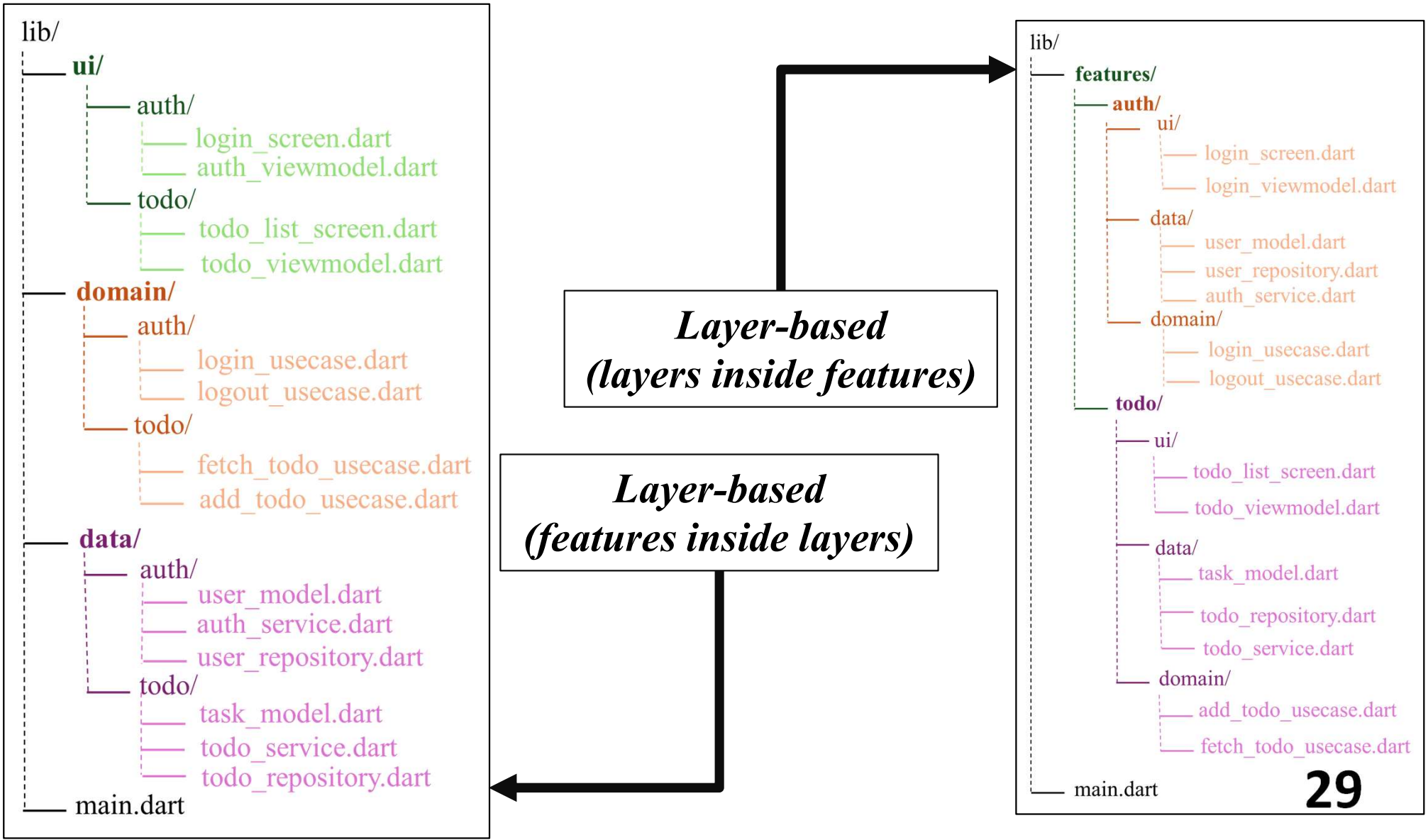
Type-based



Feature-based

Software architecture in Flutter

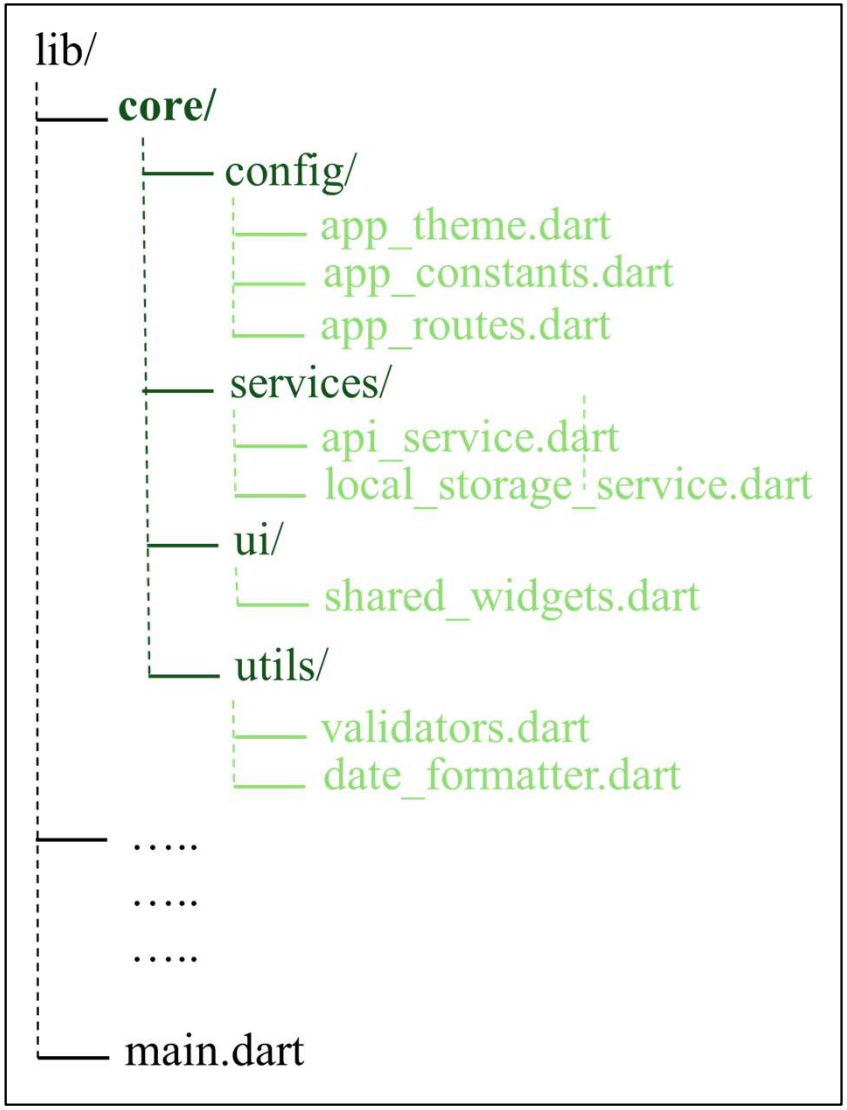
- **Common architectural patterns: project structure.**



Software architecture in Flutter

- **Common architectural patterns: project structure.**

Shared files structure



Software architecture in Flutter

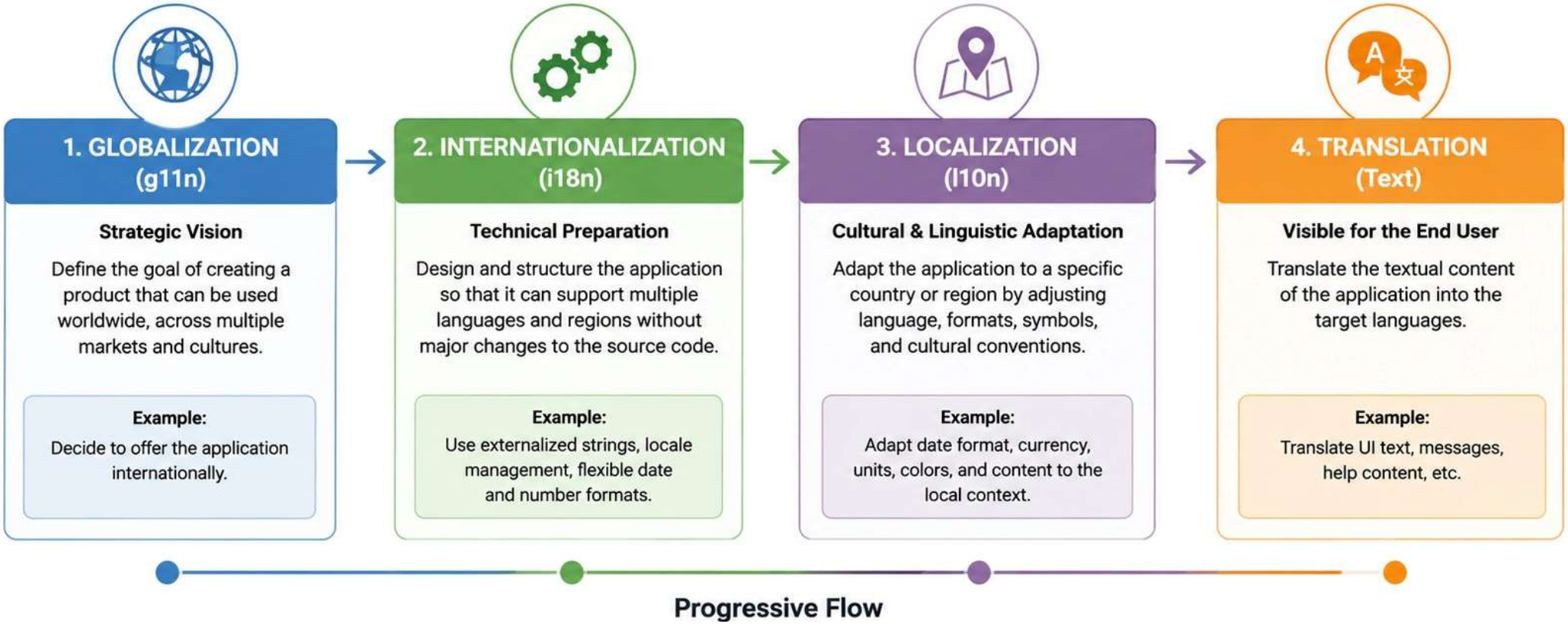
- **Summary:**

- Architectural patterns apply software design principles to build Flutter applications that are **maintainable**, **scalable**, **testable**, and **reusable**.
- **MVC** separates applications into **Model**, **View**, and **Controller**.
- **MVVM** introduces a **ViewModel** to manage UI logic and state
- **Clean Architecture** organizes applications into independent layers with clear responsibilities and enforces the **Dependency rule**.
- The choice of architecture depends on the application **complexity** and **requirements**.

Plan

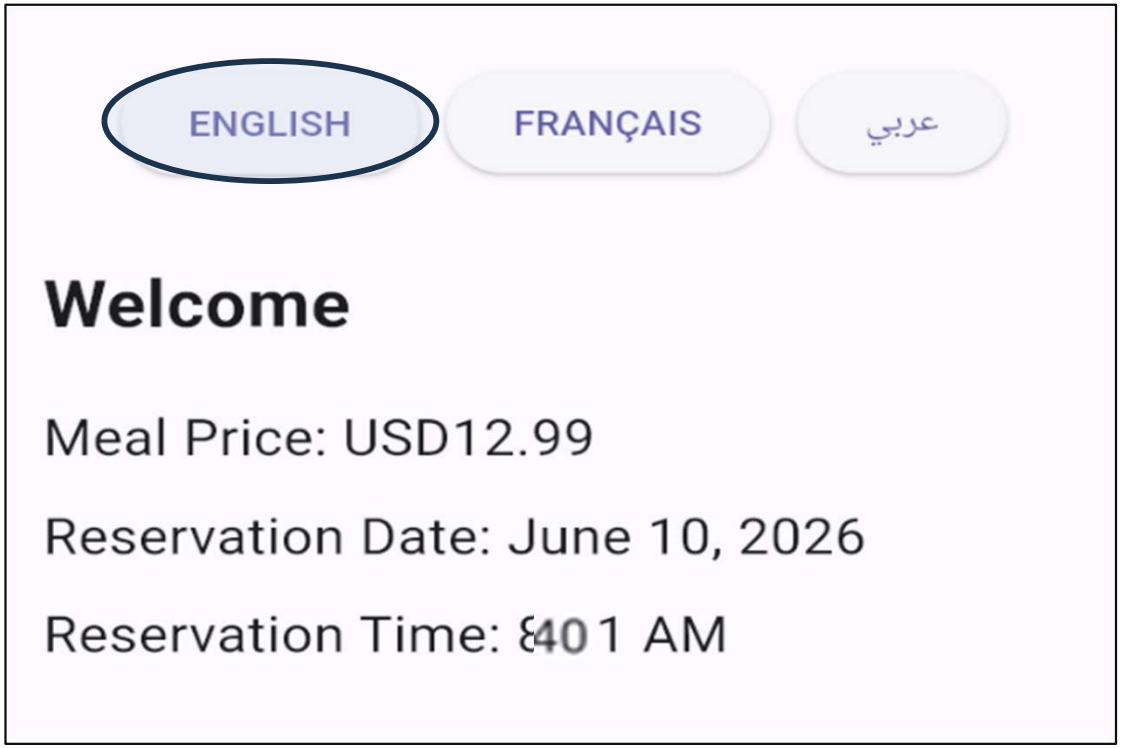
1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC.**
     - **MVVM.**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Globalization of Flutter apps



Globalization of Flutter apps

- **Meal reservation example:**



Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC.**
     - **MVVM.**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

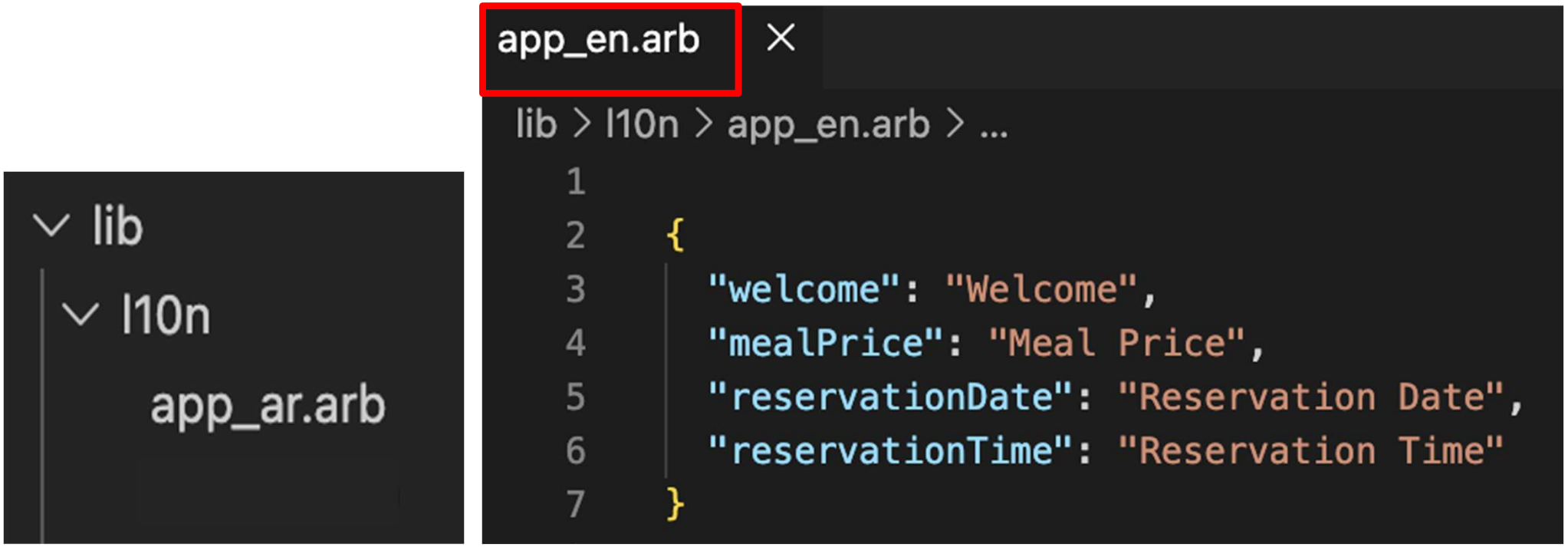
Globalization of Flutter apps

- **Internationalization:** add *flutter_localizations* dependency.

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

Globalization of Flutter apps

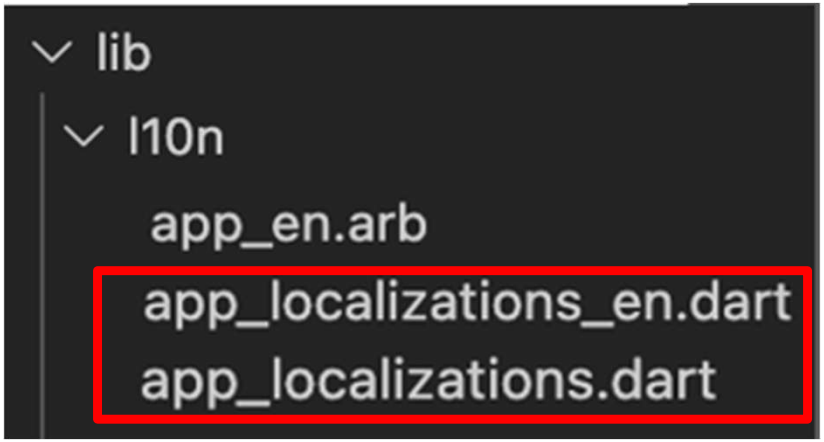
- **Internationalization:** externalize application texts.



Globalization of Flutter apps

- **Internationalization:** generate the localization class using the command *"flutter gen-l10n"*.

```
flutter:
  uses-material-design: true
  generate: true
```




```
import 'l10n/app_localizations.dart';
```

Globalization of Flutter apps

- **Internationalization:** configure the app for multilingual support.

```
MaterialApp( localizationsDelegates: AppLocalizations.localizationsDelegates,
             supportedLocales: AppLocalizations.supportedLocales, )
```

➢ Explicitly define supported locales:

```
MaterialApp( localizationsDelegates: AppLocalizations.localizationsDelegates,
             supportedLocales: const [Locale(' en '),
                                      Locale(' ar '),
                                      Locale(' fr')], )
```

Globalization of Flutter apps

- **Internationalization:** use localized strings in the UI.

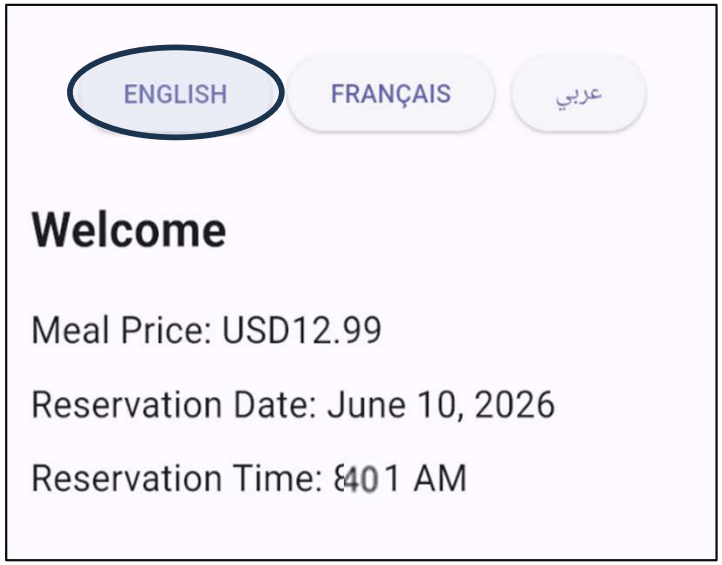
```
Text(AppLocalizations.of(context)!.welcome),
Text(AppLocalizations.of(context)!.mealPrice),
Text(AppLocalizations.of(context)!.reservationDate),
Text(AppLocalizations.of(context)!.reservationTime),
```

Plan

1. **Software architecture in Flutter:**
   - Problem & Context.
   - Design principles.
   - Common architectural patterns:
     - **MVC.**
     - **MVVM.**
     - **Clean Architecture.**
   - Project structure.
2. **Globalization of Flutter apps:**
   - Internationalization.
   - Localization.

Globalization of Flutter apps

- **Localization:** adapt the app to different languages.



Globalization of Flutter apps

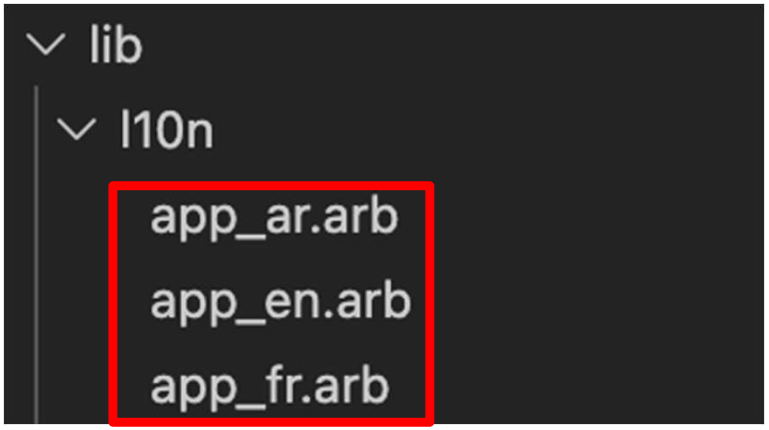
- **Localization:** add *intl* dependency.

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
  intl: any
```

```
import 'package:intl/intl.dart';
```

Globalization of Flutter apps

- **Localization:** provide translations for each supported language.



app_fr.arb

lib > l10n > app_fr.arb > ...

```
{
  "welcome": "Bienvenue",
  "mealPrice": "Prix du repas",
  "reservationDate": "Date de réservation",
  "reservationTime": "Heure de réservation"
}
```

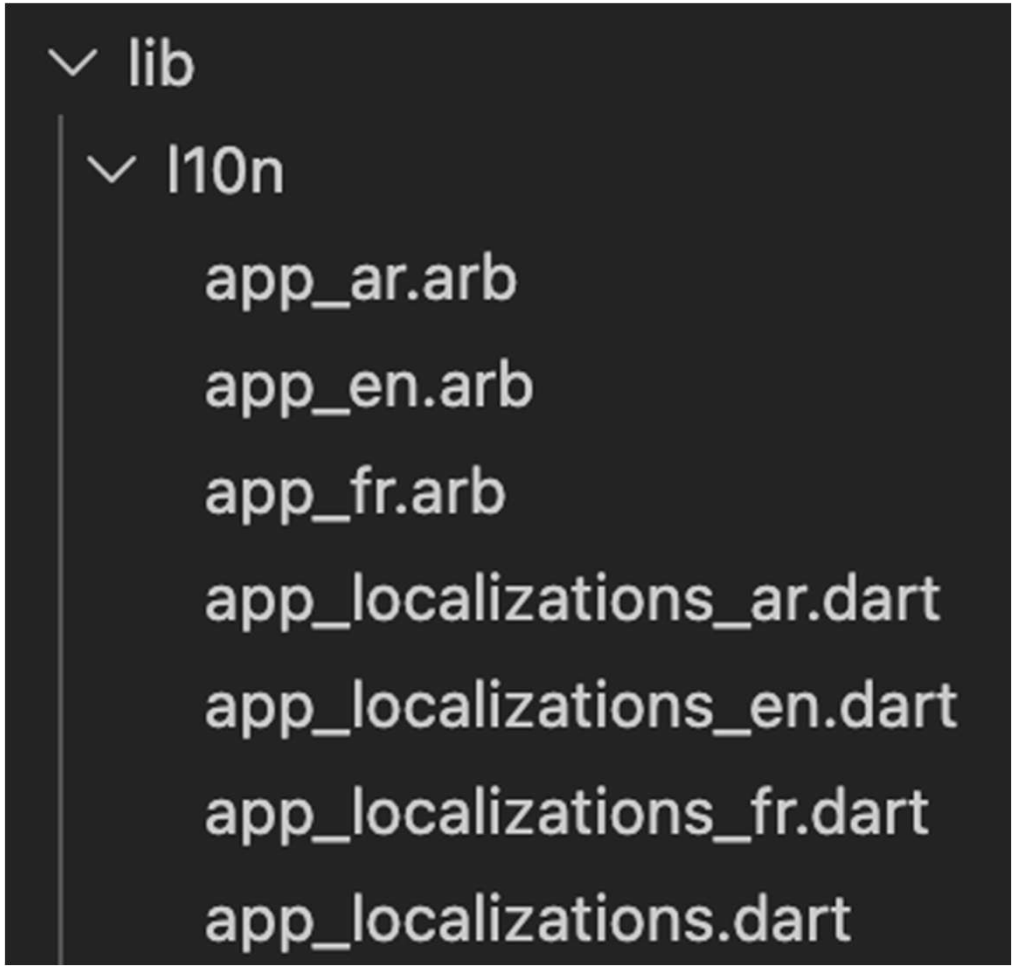
app_ar.arb

lib > l10n > app_ar.arb > ...

```
{
  "welcome": "مرحبا",
  "mealPrice": "سعر الوجبة",
  "reservationDate": "تاريخ الحجز",
  "reservationTime": "وقت الحجز"
}
```

Globalization of Flutter apps

- **Localization:** regenerate the *AppLocalizations* classes.



Globalization of Flutter apps

- **Localization:** test the application with different locales.

```
MaterialApp ( localizationsDelegates: AppLocalizations.localizationsDelegates,
              supportedLocales: AppLocalizations.supportedLocales,
              locale: current_locale ,
              ……..
            );
```

```
onPressed: () {
      setState(() { current_locale = Locale (' fr');});
}
```

Globalization of Flutter apps

- **Localization:** adapt locale-specific formats (**date**, **time**, **currency**, **number**, … etc.).

```
DateFormat.yMd( Localizations.localeOf(context).toString( ),
).format(DateTime (2026, 6, 21 ));
```

```
DateFormat.jm(Localizations.localeOf(context).toString(),
).format(DateTime.now());
```

```
NumberFormat.decimalPattern( Localizations.localeOf(context).toString(),
).format(1000.5);
```

```
NumberFormat.simpleCurrency(Localizations.localeOf(context).toString(),
).format(10);
```

Globalization of Flutter apps

- **Localization:** adapt locale-specific formats (**date**, **time**, **currency**, **number**, … etc.).

```
double temperatureC = 25;
String locale = Localizations.localeOf(context).countryCode ?? " ;
double temperature;
String unit;
        if (locale == 'US') {temperature = (temperatureC * 9 / 5) + 32;
                             unit = '°F'; }
        else { temperature = temperatureC;
               unit = '°C'; }
```

```
double distanceKm = 10;
String locale = Localizations.localeOf(context).countryCode ?? '';
double distance;
String unit;
        if (locale == 'US') {distance = distanceKm * 0.621371;
                             unit = 'mi'; }
        else { distance = distanceKm;
               unit = 'km'; }
```

Conclusion

- Well-designed architecture makes Flutter applications easier to **scale**, **test**, and **maintain** over time.
- Different architectural patterns can be used depending on the project complexity:
  - *MVC* → simple applications.
  - *MVVM* → better state management.
  - *Clean Architecture* → large and complex systems.
- Architecture provides a clear structure for the codebase, making **team collaboration** easier.
- Globalization **(i18n & l10n)** allows applications to be adapted for multiple *languages* and *locales*.

Appendix: To-Do app

- **Model:**

task_model.dart

```dart
// Definition of the Task data model
class Task {
final String id;
final String title;
final String description;
final DateTime deadline;
final String status;
// Constructor
Task({ required this.id,
       required this.title,
       required this.description,
       required this.deadline,
       required this.status,
});
// Method to create a Task object from a JSON map (i.e. deserialization).
// factory is used to return a new instance of Task from JSON map.
factory Task.fromJson(Map<String, dynamic> json)
       => Task( id: json['id'],
                title: json['title'],
                description: json['description'],
                deadline: DateTime.parse(json['deadline']), // Convert string to DateTime
                status: json['status'],);
// Method to convert a Task object into a JSON map (i.e. serialization).
Map<String, dynamic> toJson() => { "id": id, "title": title, "description": description,
                    "deadline": deadline.toIso8601String(), // Convert DateTime to string
                    "status": status,};
}
```

Appendix: To-Do app

- **Data source (services) :** local.

local_task_datasource.dart

```dart
class LocalTaskDataSource {
// Fetch all tasks from SharedPreferences
Future<List<Task>> fetchTasks() async {
      final prefs = await SharedPreferences.getInstance();
      final String? tasksJson = prefs.getString("tasks");
      if (tasksJson == null) return [];
      final List jsonTasksList = json.decode(tasksJson);
      return jsonTasksList.map((e) => Task.fromJson(e)).toList();
       }
// Add a new task to SharedPreferences
Future<void> addTask(Task task) {
    final prefs = await SharedPreferences.getInstance();
    final tasksList = await fetchTasks();
    tasksList.add(task);
    final  String tasksJson = json.encode(tasksList.map((e) => e.toJson()).toList());
    await prefs.setString("tasks", tasksJson);
    }
// Delete a task from SharedPreferences
 Future<void> deleteTask(String id) async {
    final prefs = await SharedPreferences.getInstance();
    final tasksList = await fetchTasks();
    tasksList.removeWhere((t) => t.id == id);
    final String tasksJson = json.encode(tasksList.map((e) => e.toJson()).toList());
    await prefs.setString("tasks", tasksJson);
    }
}
```

Appendix: To-Do app

- **Data source (services) :** remote.

remote_task_datasource.dart

```dart
// RemoteTaskDataSource manages access to tasks stored on a remote server (API REST).
class RemoteTaskDataSource {

final String baseUrl; // Base URL of the remote API
RemoteTaskDataSource(this.baseUrl); // Constructor

// Fetch the list of tasks stored on the remote server
Future<List<Task>> fetchTasks() async {
final response = await http.get(Uri.parse('$baseUrl/tasks'));
      if (response.statusCode == 200) {
            final data = json.decode(response.body) as List;
            return data.map((e) => Task.fromJson(e)).toList();}
      throw Exception("Failed to load tasks from server");
    }
// Add a new task to the remote server
Future<void> addTask(Task task) async {
final response = await http.post(Uri.parse('$baseUrl/tasks'),
      body: json.encode(task.toJson()), // Convert Task object to JSON
      headers: {"Content-Type": "application/json"});
      if (response.statusCode != 201) {
            throw Exception("Failed to add task to server");}
}
// Delete a task from the remote server
Future<void> deleteTask (String id) async {
final response = await http.delete(Uri.parse('$baseUrl/tasks/$id'));
      if (response.statusCode != 200 && response.statusCode != 204) {
            throw Exception("Failed to delete task from server");}
  }
}
```

Appendix: To-Do app

- **Repository & repository implementation:**

task_repository_impl.dart

```dart
// Repository implementation (uses local + remote)

class TaskRepositoryImpl implements TaskRepository {
final LocalTaskDataSource localDataSource;
final RemoteTaskDataSource remoteDataSource;
// Constructor
TaskRepositoryImpl({ required this.localDataSource,
                     required this.remoteDataSource,});
// Implementation of the getTasks method
@override
Future<List<Task>> getTasks() async {
try {// Try to fetch from remote first
     final tasks = await remoteDataSource.fetchTasks();
     return tasks;
  } catch (e) {// If remote fails, fallback to local
     return await localDataSource.fetchTasks();
     }
}
// Implementation of the addTask method
@override
Future<void> addTask(Task task) async {
try {
     await remoteDataSource.addTask(task);
     await localDataSource.addTask(task);
  } catch (e) {
      throw Exception("Failed to add task: $e");
  }
}
// Implementation of the deleteTask method
@override
Future<void> deleteTask(String id) async {
try {
     await remoteDataSource.deleteTask(id);
     await localDataSource.deleteTask(id);
  } catch (e) {
     throw Exception("Failed to delete task: $e");
  }
 }
}
```

task_repository.dart

```dart
// Task repository interface (abstract contract)
abstract class TaskRepository {
  Future <List<Task>> getTasks();
  Future <void> addTask(Task task);
  Future <void> deleteTask(String id);
}
```

Appendix: To-Do app

- **Use cases:**

add_task.dart

```
class AddTask {
final TaskRepository repository;

AddTask(this.repository);
Future<void> call(Task task) async {
 return await repository.addTask(task);
 }
}
```

delete_task.dart

```
class DeleteTask {
final TaskRepository repository;

DeleteTask (this.repository);
Future<void> call(String id) async {
return await repository.deleteTask(id);
  }
}
```

get_tasks.dart

```
class GetTasks {
final TaskRepository repository;

GetTasks(this.repository);
Future<List<Task>> call() async {
return await repository.getTasks();
 }
}
```

Appendix: To-Do app

- **Entity:**

task_entity.dart

```dart
// Definition of the Task domain model (entity)
class TaskEntity {
final String id;
final String title;
// Constructor
TaskEntity ({required this.id, required this.title,});
}
```

Appendix: To-Do app

- **ViewModel:**

task_viewmodel.dart

```dart
// ChangeNotifier is used as a state management approach that
allows managing and propagating state changes
class TaskViewModel extends ChangeNotifier {
// Use cases are injected into the ViewModel
final GetTasks getTasksUseCase;
final AddTask addTaskUseCase;
final DeleteTask deleteTaskUseCase;
// Private list of tasks managed by the ViewModel
List<Task> _tasks = [];
// Public getter so the UI can access the tasks
List<Task> get tasks => _tasks;
// Constructor
TaskViewModel({
required this.getTasksUseCase,
required this.addTaskUseCase,
required this.deleteTaskUseCase,
});
// Loads the list of tasks by invoking the GetTasks use case.
Future<void> loadTasks() async {
      _tasks = await getTasksUseCase();
       notifyListeners();
  }
// Adds a new task by invoking the AddTask use case,
// then refreshes the list of tasks
Future<void> addTask(Task task) async {
        await addTaskUseCase(task);
        await loadTasks(); // refresh
  }
// Deletes a task by invoking the DeleteTask use case,
// then refreshes the list of tasks
Future<void> deleteTask(String id) async {
      await deleteTaskUseCase(id);
        await loadTasks(); // refresh
  }
}
```

Appendix: To-Do app

- **View:**

task_view.dart

```
lass TaskView extends StatelessWidget {
onst TaskView({super.key});

verride
idget build(BuildContext context) {
eturn Scaffold(
    appBar: AppBar(title: const Text("To-do tasks app")),
    body: Consumer<TaskViewModel>(
    builder: (context, viewModel, child) {
            final tasks = viewModel.tasks;
            return ListView.builder(
                    itemCount: tasks.length,
                    itemBuilder: (context, index) {
                          final task = tasks[index];
                          return ListTile(
                                title: Text(task.title),
                                trailing: IconButton(
                                        icon: Icon(Icons.delete),
                                            onPressed: () {viewModel.deleteTask(task.id);},
                                ), ); }, ); }, ),
     floatingActionButton: FloatingActionButton(
                               onPressed: () {
                                   context.read<TaskViewModel>().addTask(
                                Task(id: DateTime.now().toString(), title: "New Task",
                                      description: '', deadline: DateTime(2025, 9, 14)),);
                                 },
                                child: Icon(Icons.add),
                              ),); }
}
```