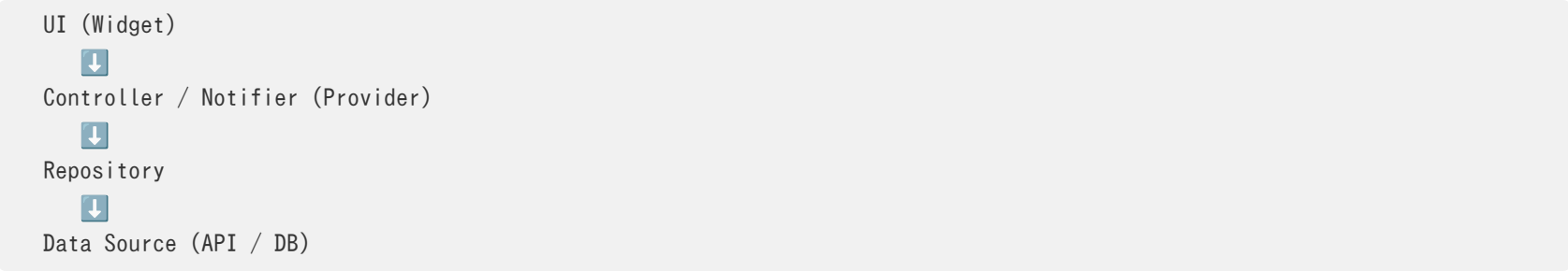


Riverpod Architecture Interaction Rules (Flutter)

Riverpod Architecture Interaction Rules (Flutter)

Think of it like this:



Rule 1: UI cannot directly talk to Data or Domain layers

- ✓ UI → Provider → Repository
- ✗ UI → Repository (skip provider)
- ✗ UI → API or DB directly

Reason:

Providers act as a “bridge” or **mediator** between UI and logic layers — keeping UI reactive, testable, and clean.

Rule 2: Each layer has only one-way access

From	Can Access	Cannot Access
UI	Provider (Notifier / StateProvider)	Repository or DataSource directly
Provider (Controller)	Repository, Domain Logic	UI Widgets or BuildContext
Repository	Data Sources (API, DB, Local Storage)	UI or Provider
Data Source	Network/Local System	Anything above it

✓ Correct Example

```
// presentation/controller/user_controller.dart
class UserController extends Notifier<UserState> {
  final UserRepository repo;
  UserController(this.repo);

  Future<void> fetchUser() async {
```

```

    final user = await repo.getUser(); // gets from data layer
    state = UserState.loaded(user);
  }
}

// ui/screen/user_screen.dart
final userProvider = NotifierProvider<UserController, UserState>(
  () => UserController(ref.read(userRepoProvider)),
);

Widget build(context, ref) {
  final state = ref.watch(userProvider);
  return state.when(
    data: (user) => Text(user.name),
    loading: () => CircularProgressIndicator(),
    error: (e, _) => Text('$e'),
  );
}

```

Rule 3: Domain layer stands between Controller and Repository (optional but recommended)

If your app has **business logic**, insert a Domain layer (use cases).

UI → Controller → UseCase (Domain) → Repository → DataSource

Each layer focuses on one concern:

- **UseCase:** Defines business actions (e.g., login, fetchUserProfile)
- **Repository:** Knows how to get data (network, local, etc.)
- **Controller:** Controls UI state and calls usecases

Example Flow

```

User taps button
↓
UIScreen → reads → UserController
↓
UserController → calls → GetUserUseCase
↓
GetUserUseCase → calls → UserRepository
↓
UserRepository → calls → ApiService
↓
ApiService → returns data → Repository → Controller → UI updates

```

Rule 4: Providers connect the layers

Each layer should be **injected** via a Riverpod provider.

```

final apiProvider = Provider((ref) => ApiService());
final userRepoProvider = Provider((ref) => UserRepository(ref.read(apiProvider)));
final getUserUseCaseProvider = Provider((ref) => GetUserUseCase(ref.read(userRepoProvider)));

```

```
final userControllerProvider =  
    NotifierProvider<UserController, UserState>(() => UserController(ref.read(getUserUseCaseProvider)));
```

This keeps **clear dependency flow** — no circular access.

Rule 5: Never reverse access

- Repository must not call Controller.
- Data Source must not know about Repository.
- UseCase must not depend on UI or Providers.
- Providers must not depend on Widgets or BuildContext.

Each layer is **downward-only** in communication.

Rule 6: Data flows downward, state flows upward

- Data (API results) flows **down** → **up** (repository → controller → UI).
- User interactions flow **up** → **down** (UI → controller → repository).

Rule 7: Provider is your single bridge

Even when multiple layers exist:

- UI communicates **only with providers**
- Providers call **repositories or usecases**
- Providers handle state updates

Visual Flow Summary

```
[ UI Layer ]  
  ↓ reads / watches  
[ Riverpod Provider (Controller / Notifier) ]  
  ↓ calls  
[ Domain / UseCase Layer (optional) ]  
  ↓ delegates to [ Repository Layer ]  
  ↓ connects to [ Data Sources (API, Local DB, Cache) ]
```