



FlutterKaigi 2024

マッチングアプリ『Omiai』の
Flutterへのリプレイスの挑戦

Kosuke Saigusa (株式会社 Omiai)

- Kosuke Saigusa (@kosukesaigusa)
- 株式会社 Omiai Flutter テックリード
- FlutterNinjas 2024, FlutterKaigi 2023 等で登壇
- FlutterGakkai, 東京 Flutter ハッカソン運営





※1 2012年のサービス開始以来の累計登録数。

※2 2012年のサービス開始以来の累計マッチング数。2024年7月時点自社調べ。



Omiaiがなければこんな素敵なお逢いがなかった
と思うとOmiaiにはすごく感謝しています



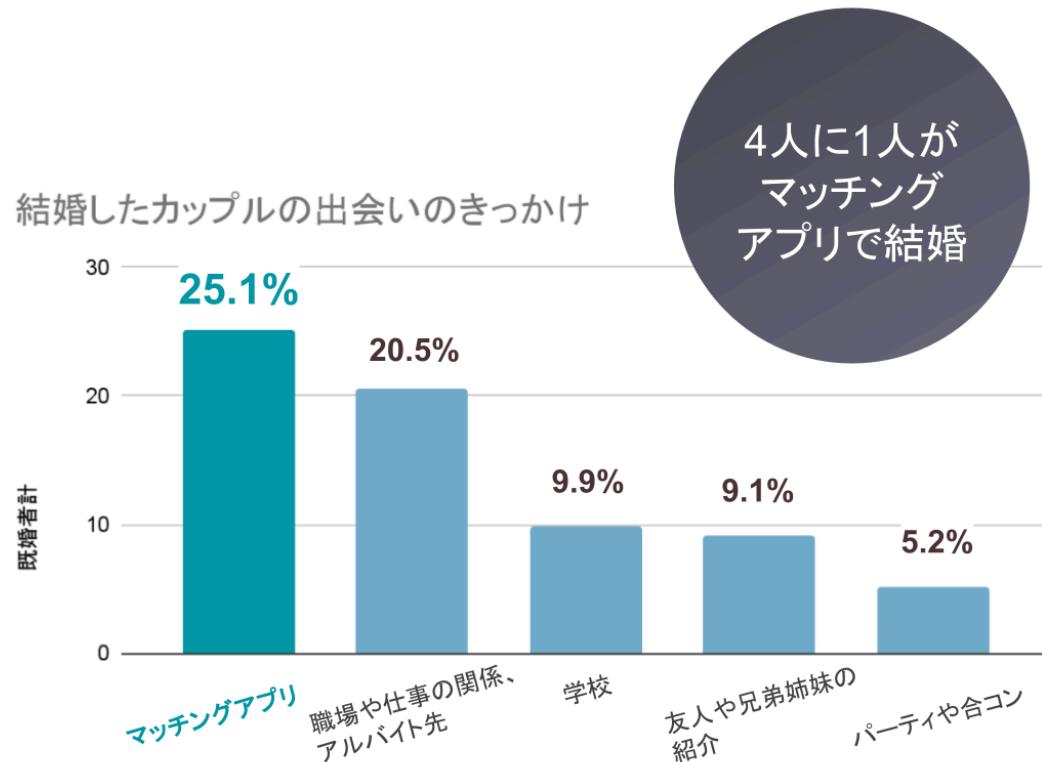
2人とも同じ時期にOmiaiに登録して、誕生日も1
日違い！まさに運命を感じています



Omiaiがなければ今の生活がないので、良縁に結
びついたことを感謝しております

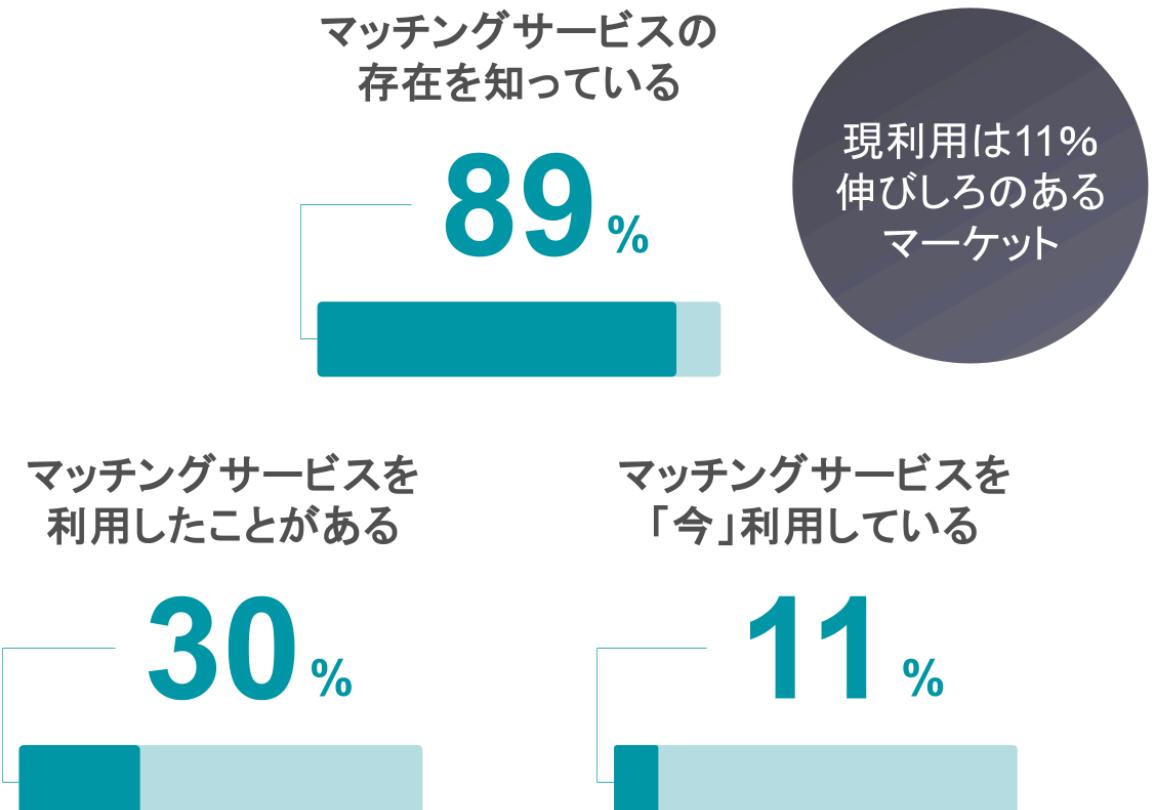
昨今のマッチングアプリと市場

Omiai



出典：こども家庭庁調べ（2024年8月26日発表ウェブアンケート）

※15～39歳の既婚者20,000人



出典：マクロミルインターネット調査（2023年5月、20～39歳独身男女8,244人）

enITO
GROUP

with



- シリアス系国内シェア2位
- 20代中心 値値観重視の恋人探し
- 2015年9月にサービス開始



- 日本初 × シリアス系国内シェア3位
- 恋愛結婚をかなえるまじめな出会い
- 2012年2月にサービス開始



経営／管理

新規事業
内部監査
Trust&Safety
広報／人事
コーポレート
社内システム



サービス開発／運営

PdM／プランナー
エンジニア
デザイナー
マーケティング
データアナリスト
カスタマーサポート
QA

本セッションのゴール

マッチングアプリ『Omiai』を題材に
大規模アプリの Flutter へのリプレイスを
成功に導くアプローチを考える

なぜ『Omiai』は Flutter にリプレイスするのか？

背景

- 2012 年から長年運営されてきたサービス
- iOS・Android (, Web) の各プラットフォームの古いコードベースが抱える多くの技術的負債や仕様差異
- サーバサイドの負債の解消・仕様変更もクライアントアプリが理由で進めづらい

目標

- Flutter へのリプレイスによる単一コードベース化と技術的負債の返済
- リプレイス中も機能追加は止めない
- toC アプリとしてあるべき素早く・細かい開発サイクルでの開発・検証を実現する

どのようにリプレイスするのか？

✓ 負債化を繰り返さず、高い開発生産性を維持し続ける

- アーキテクチャ、CI、テスト、ドキュメンテーションなどで多くの工夫

✓ Flutter 経験の無いメンバーや新規参画メンバーが早期に活躍できる

- Flutter 経験者ゼロ・業務委託中心組織から正社員採用によるチームの拡大へ

✓ リプレイス中も機能追加は止めない

- Add-to-App を利用して既存アプリから Flutter をモジュールとして呼び出す
- 主要なユーザー体験から順に、新しい機能を追加しながら Flutter にリプレイスする

陥りがちな現実

```
ElevatedButton(  
    onPressed: () async {  
        await someRepositoryMethod();  
    },  
    /** 省略 */  
);
```

- UI 層から直接 Repository のメソッドを呼び出すのを禁止している場合
- コード規約で縛り、PR レビューで防ぐしか方法がない
- 見逃されたコードが生き残ると、コード規約が曖昧に

```
import 'package:flutter/material.dart';
import 'package:flutter_localizations/flutter_localizations.dart';

Future<void> someLogic(BuildContext context) async {
  await doSomething();
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text(AppLocalizations.of(context).success)),
  );
}
```

- 業務ロジックの実装に `BuildContext` が依存する
- ロジックのユニットテストが Dart のユニットテストで書けない

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

ElevatedButton(
  onPressed: () async {
    try {
      await doSomething();
    } on DioException catch (e) {
      /** 省略 */
    },
  },
  /** 省略 */
),
```

- UI 層が特定の HTTP クライアントパッケージに依存するのは不自然（知るべきでない）

```
ElevatedButton(  
    onPressed: () async {  
        try {  
            await doSomething();  
        } on APIException catch (e) {  
            await showDialog<void>(  
                context: context,  
                builder: (context) => AlertDialog(content: Text('${e.statusCode}')),  
            );  
        },  
    },  
    /** 省略 */  
,
```

- UI 層で HTTP 通信が起因の例外を捕捉するのは OK ?
- 捕捉した例外をどのようにユーザー体験に反映するかの方針が明確でない
- 図らずエラーコードなどの情報をユーザーに見せてしまう

```
class SomeRepository {  
    /** 省略 */  
  
    Future<Something> fetchSomething({/** 省略 */}) async {  
        final accessToken = _ref.read(accessTokenProvider);  
        final response = await _client.request({/** 省略 */});  
        /** 省略 */  
    }  
}
```

- 通信層が、業務知識としての認証情報に依存する
- 通信層のユニットテストを書くのに、業務知識をモックする必要がある
- もしこの依存が許されるなら、あらゆる業務知識に同様に依存し始める危険がある

```
class PaginatedFetcher<T> {  
    /** 省略 */  
  
    Future<List<T>> fetchNextPage(int page, int perPage) async {  
        /** 省略 */  
        final items = await fetch(**省略**);  
        /** 省略 */  
        await ref.read(someUseCaseProvider).doSomething();  
        return items;  
    }  
}
```

- 当初の汎用的な基盤実装が、いつの間にか意図しない対象に依存するようになる
- 汎用性は失われ、責務が曖昧な負債コードに

```
ElevatedButton(  
    onPressed: () async {  
        try {  
            await doSomething();  
        } on APIException catch (e) {  
            await showDialog<void>(  
                context: context,  
                builder: (context) => AlertDialog(content: Text(e.message)),  
            );  
        },  
        /** 省略 */  
    ),
```

- ユーザーに知らせるエラーメッセージに、サーバサイドからのレスポンスデータがそのまま利用される
- クライアントアプリとサーバサイドアプリの境界が曖昧

どのように防ぐ？

可能な限り仕組みで防ぎたい！

✓ マルチパッケージ構成

- レイヤーごとにパッケージを分ける
- 各パッケージの責務と許可される依存を厳密にする
- Flutter の世界と Dart の世界、クライアントアプリとサーバサイドアプリの境界を明確にする

✓ ユニットテストを十分に継続的に書く

- 業務ロジック層およびそれより抽象的な層で、ユニットテストがカバレッジ 100% で記述できる

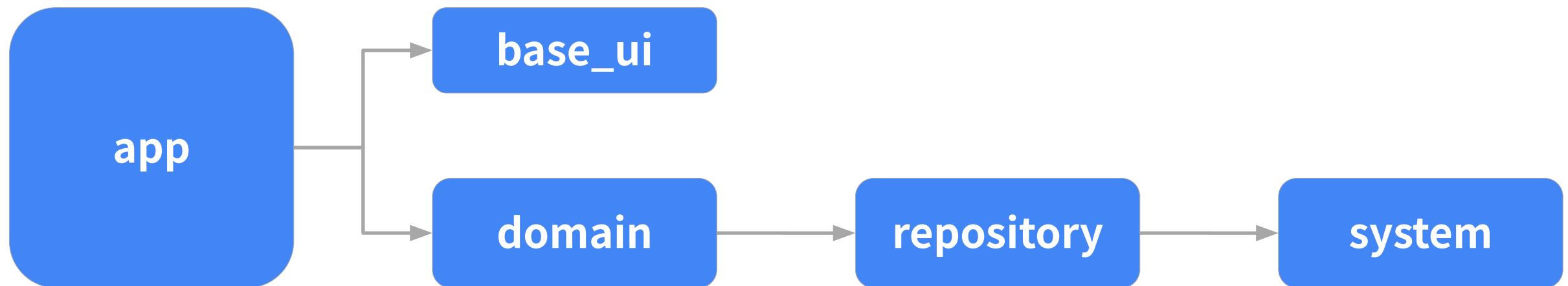
✓ 実装を可能な限りパターン化する

- 典型的な実装をパターン化し、新規参画メンバーの早期の活躍を実現する

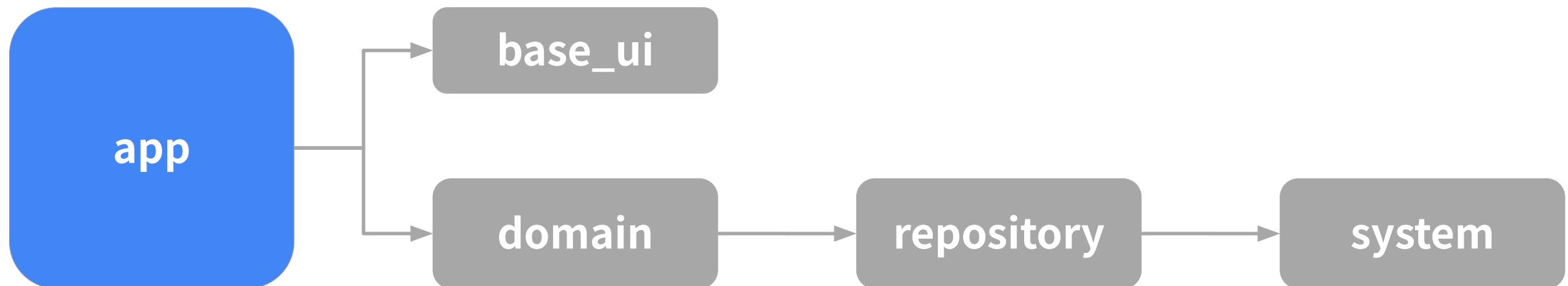
パッケージ構成

- レイヤーごとにパッケージを分ける
- 各パッケージに許される直接的な依存は矢印の先のパッケージのみに限定される
- 他にも `Unimplemented` なインターフェースのみを定義し、各パッケージに利用させるパッケージなどもある

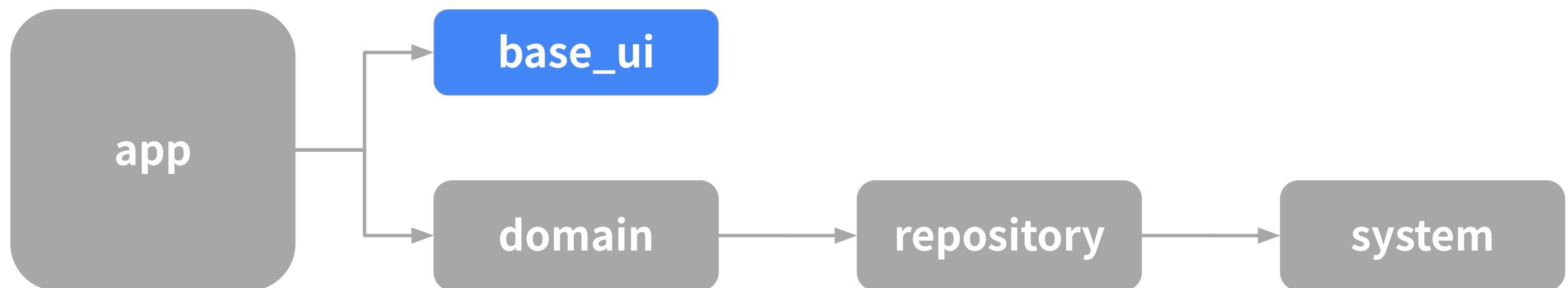
※ ことば遣いは Flutter 公式の "Architecting Flutter apps" や Android Developers の "Guide to app architecture" に類似



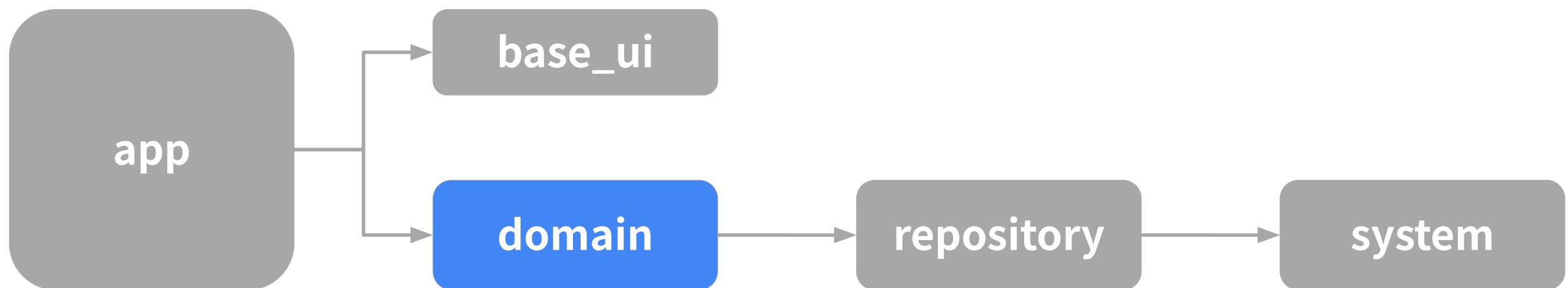
- Flutter アプリのエントリポイントとしての実装
 - `ProviderScope.overrides` による依存の注入
- UI 層としてのユーザー体験を実装
 - ウィジェットツリーの構築、画面遷移、各画面やコンポーネントの描画とインターラクション



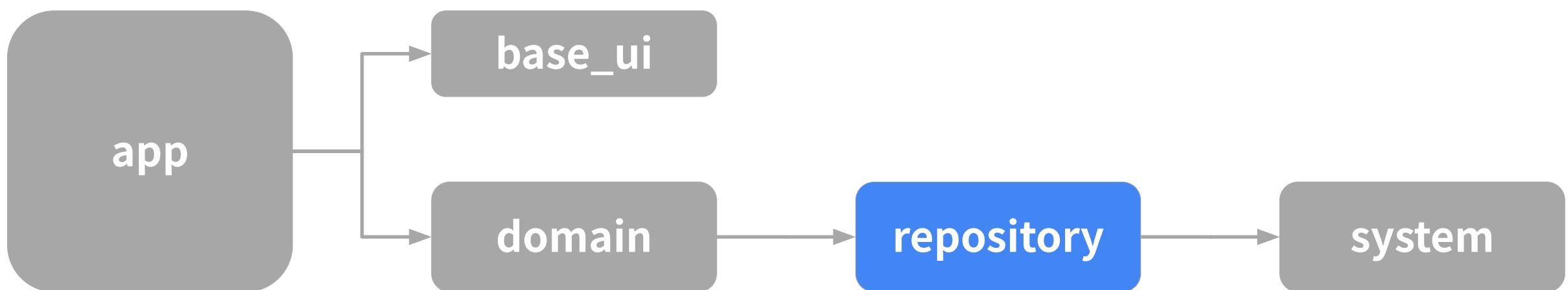
- テキストスタイル、色などのスタイルガイドの実装
- 画像、ボタン、AppBar、TextField などの基礎的な UI 部品の実装



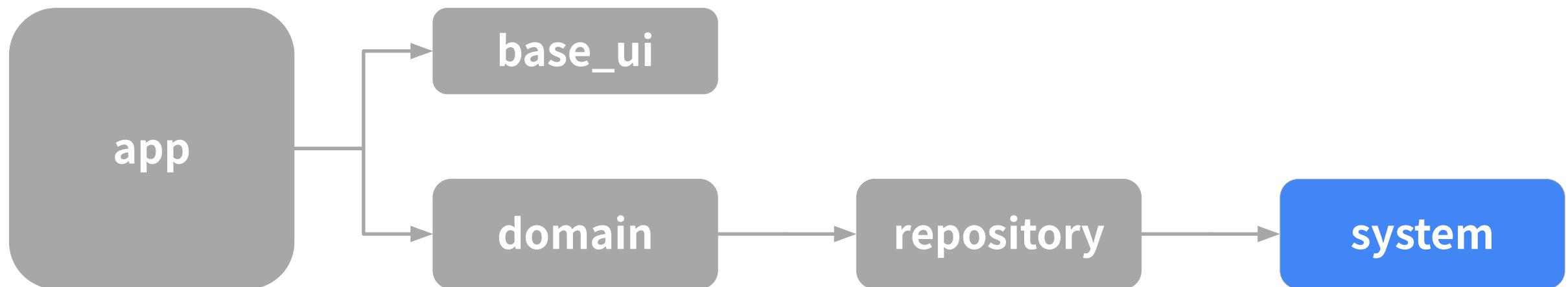
- クライアントアプリの業務概念を表すエンティティの定義
- ユーザー体験として利用する例外型の定義
- データの取得や保存などを含む業務ロジックの記述



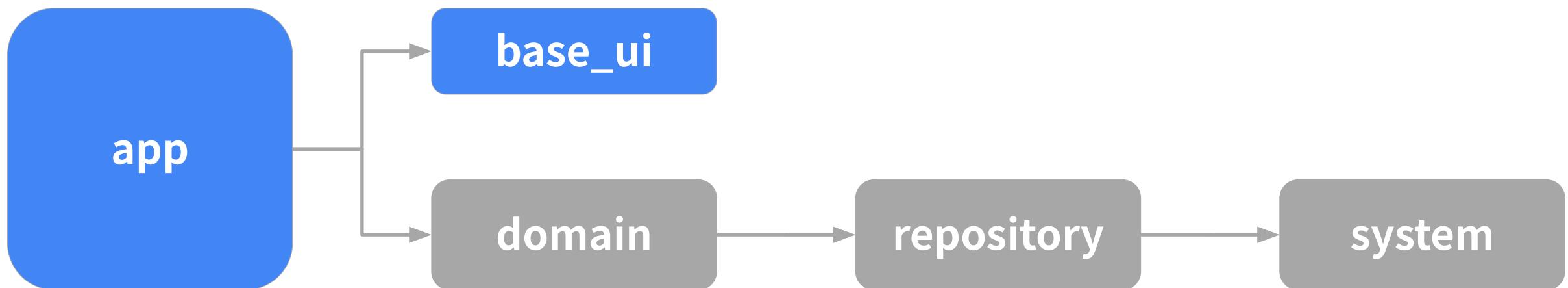
- データソースとのやり取りの実装
- 接続先のデータソースの情報は表出させない



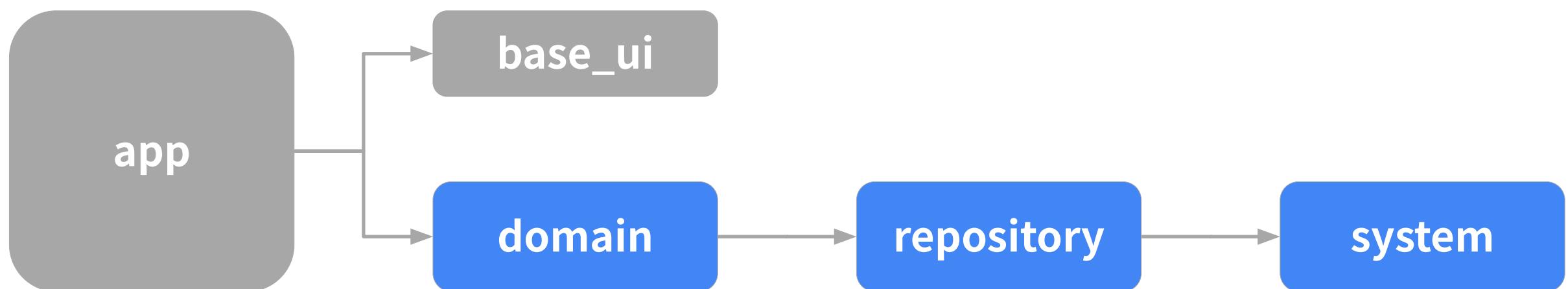
- 3rd パーティのツールをラップして基礎的な実装を記述
 - 例：HTTP クライアント、Shared Preferences, Firebase Analytics など
- 3rd パーティのツールとの依存は例外も含めて閉じ込めて外部に表出させない
- 不要な機能のインターフェースは塞いでシンプルにする



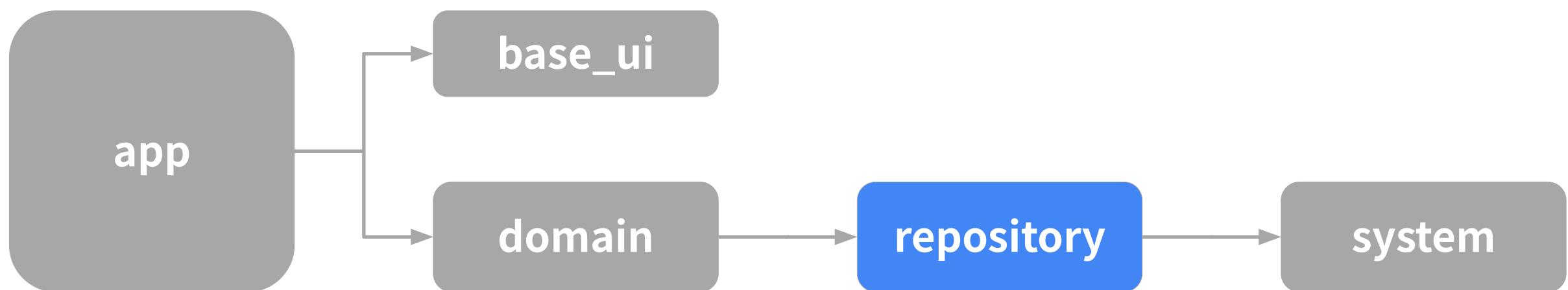
- app, base_ui パッケージは Flutter に直接依存する
- Flutter エンジニアとしてコードを書く



- domain, repository, system パッケージは Flutter には直接依存しない
- Dart エンジニアとしてコードを書く
 - Dart 言語と riverpod や freezed にさえ慣れれば Flutter 経験が浅くても書ける

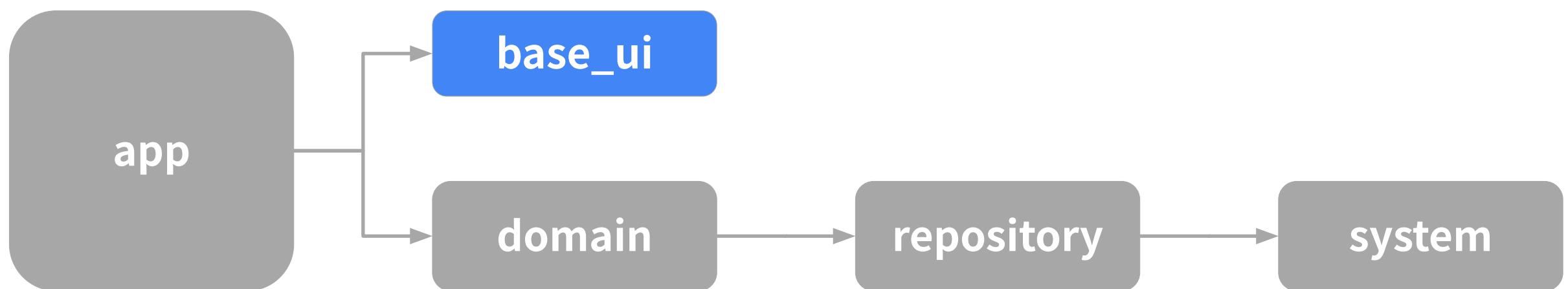


- repository パッケージがサーバサイドとの境界として働く
- サーバサイドの負債は repository パッケージで吸収し切り、app, domain に流入させない



各パッケージの実装内容

- テキストスタイル、色などのスタイルガイドの実装
- 画像、ボタン、AppBar、TextField などの基礎的な UI 部品の実装



base_ui が依存するパッケージの例

Omiqai

```
dependencies:  
  extended_image:  
  flutter:  
    sdk: flutter  
  flutter_svg:  
  
dev_dependencies:  
  flutter_gen_runner:
```

- Flutter への依存: direct
- 依存パッケージの例：
 - extended_image: 画像の基盤実装
 - flutter_gen: 静的アセットのコード生成

```
/// アプリで使用する色一覧。
abstract interface class AppColor {
    /// プライマリーの 01 の赤色。
    static const Color primary01 = Color(0xFFFF3159);

    /// プライマリーの 02 の青色。
    static const Color primary02 = Color(0xFF3F9AD1);

    /// セカンダリーの 01 のピンク色。
    static const Color secondary01 = Color(0xFFFFE8D8B);

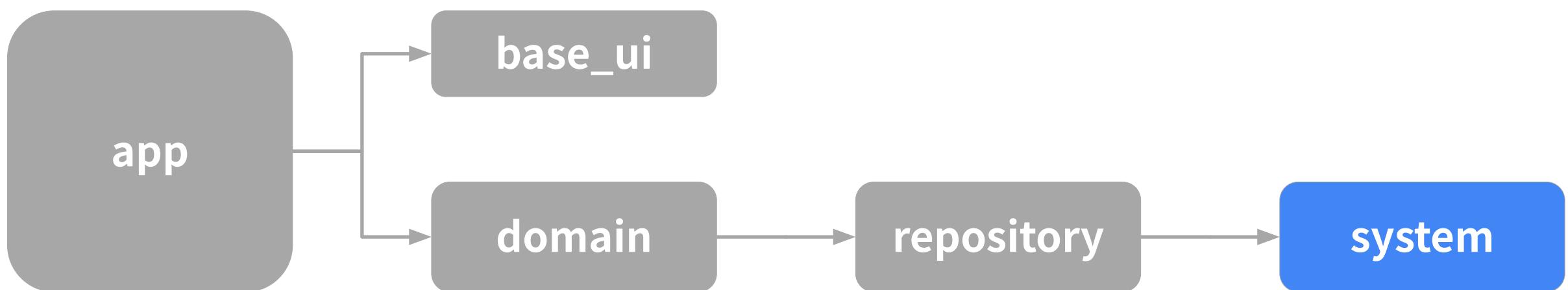
    /** 省略 */
}
```

- Figma と同名で定義された TextStyle, Color などのスタイルガイド

```
/// アプリ内で共通で用いられる [ElevatedButton].  
class CommonElevatedButton extends StatelessWidget {  
    /// S サイズの [CommonElevatedButton] を生成する。  
    const CommonElevatedButton.s({/** 省略 */});  
  
    /// S サイズのアイコン付きの [CommonElevatedButton] を生成する。  
    const CommonElevatedButton.sWithIcon({/** 省略 */});  
  
    /** 省略 */  
}
```

- ボタン、画像、AppBar、TextField などの基礎的な UI 部品
- Figma のコンポーネント定義と一致するインターフェース

- 3rd パーティのツールをラップして基礎的な実装を記述
- 外部パッケージへの依存を閉じ込めて、外部に表出させない



system が依存するパッケージの例

Omiai

```
dependencies:  
  dio:  
  firebase_analytics:  
  firebase_remote_config:  
  shared_preferences:
```

- Flutter への依存: transitive
- 各種の 3rd パーティのツールに依存する

```
class HttpClient {  
    HttpClient({required String baseUrl, Map<String, dynamic>? headers})  
        : _client = DioHttpClient(baseUrl: baseUrl, headers: headers);  
  
    Future<HttpResponse<dynamic>> request(/** 省略 **/) async {/** 省略 **/}  
}
```

- **HTTPClient** クラスの例
- 内部で dio パッケージへ依存

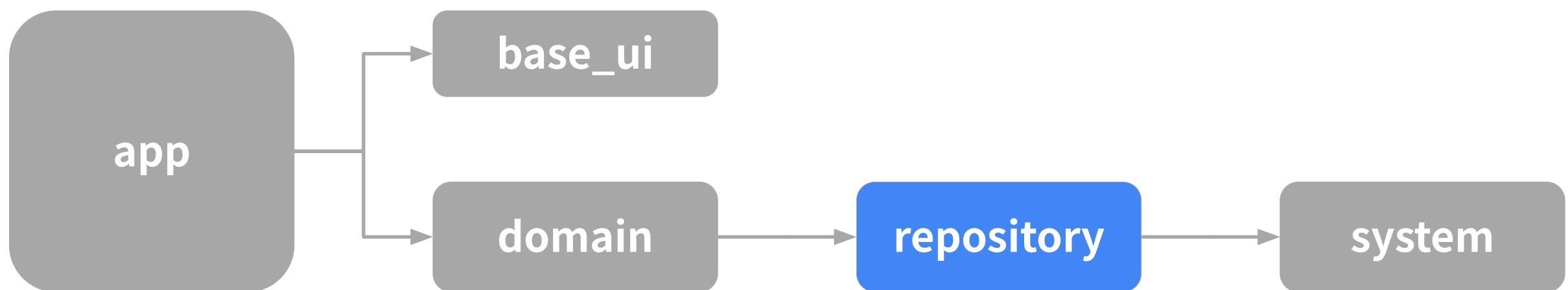
```
@freezed  
sealed class HttpResponse<T> with _$HttpResponse<T> {  
  const factory HttpResponse.success({  
    required T data,  
    required Map<String, List<String>> headers,  
  }) = SuccessHttpResponse<T>;  
  
  const factory HttpResponse.failure({  
    T? data,  
    required Object e,  
    required ErrorStatus status,  
  }) = FailureHttpResponse<T>;  
}
```

- HTTP リクエストの結果を表現する `HttpResponse` クラスの例
- `Result` 型のような形で HTTP リクエストの成功・失敗を表現

```
class HttpClient {  
    /** 省略 */  
  
    Future<HttpResponse<dynamic>> request({/** 省略 */}) async {  
        try {  
            final response = await _client.request<dynamic>({/** 省略 */});  
            return HttpResponse.success({/** 省略 */});  
        } on DioException catch (e) {  
            /** 省略 */  
            return HttpResponse.failure({/** 省略 */});  
        }  
    }  
}
```

- HTTP リクエストの結果として `HttpResponse` を返す
- dio への依存は例外も含めて外部には表出させない

- データソースとのやり取りの実装
- 接続先のデータソースの情報は表出させない



repository が依存するパッケージ

Omiq

```
dependencies:  
  freezed_annotation:  
  json_annotation:  
  riverpod:  
  system:  
    path: ../../system
```

```
dev_dependencies:  
  freezed:  
  json_serializable:  
  riverpod_generator:
```

- Flutter への依存: transitive
- system パッケージに依存して、HTTP クライアントなどを利用する
- json_serializable, freezed で Dto の定義を行う

```
@freezed  
sealed class RepositoryResult<T> with _$RepositoryResult<T> {  
  const factory RepositoryResult.success(T data) = SuccessRepositoryResult<T>;  
  
  const factory RepositoryResult.failure(  
    Object e, {  
      FailureRepositoryResultReason? reason,  
      ErrorDto? errorDto,  
    }) = FailureRepositoryResult;  
}
```

- Repository による通信結果を表現する `RepositoryResult` クラスの例
- Result 型のような形で成功・失敗を表現

repository の実装例

Omiq

```
class AccountDetailRepository {  
  /** 省略 */  
  
  Future<RepositoryResult<AccountDetailDto>> fetchAccountDetail() async {  
    final response = await _ref.read(httpClientProvider).request(** 省略 **);  
    switch (response) {  
      case SuccessHttpResponse(:final data):  
        final dto = AccountDetailDto.fromJson(data as Map<String, dynamic>);  
        return RepositoryResult.success(dto);  
      case FailureHttpResponse(** 省略 **):  
        /** 省略 */  
        return RepositoryResult.failure(** 省略 **);  
    }  
  }  
}
```

- `HttpResponse` の成功・失敗を `switch` 文で分岐して、`RepositoryResult` を返す

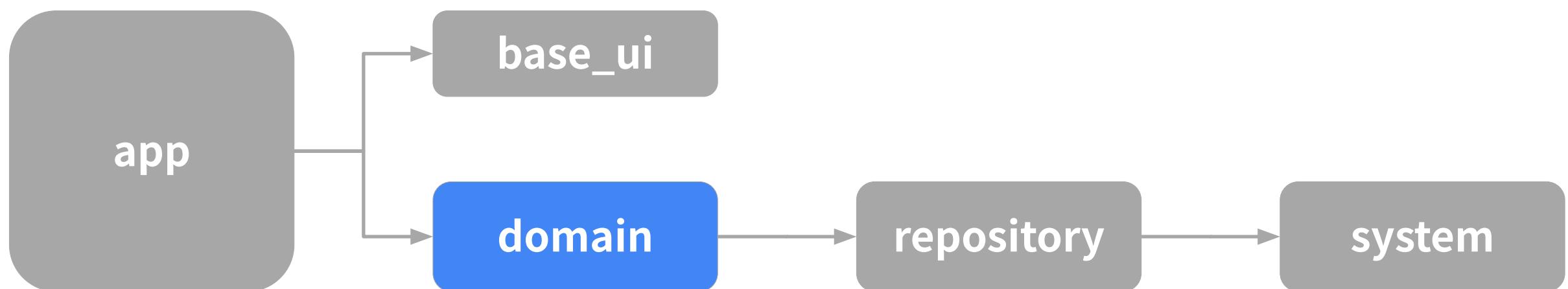
repository の実装例

Omiq

```
@freezed  
class FooDto with _$FooDto {  
  const factory FooDto({  
    @JsonProperty(name: 'inappropriate_field_name') required String someFieldName,  
    @Default([]) List<String> things,  
    @flexibleBoolConverter required bool isSomething,  
    @flexibleDateTimeConverter required DateTime someDateTime,  
  }) = _FooDto;  
  
  factory FooDto.fromJson(Map<String, dynamic> json) => _$FooDtoFromJson(json);  
}
```

- Dto として HTTP レスポンスの型定義をする
- 不適切なフィールド名や型を変換して、サーバサイドの負債を持ち込まない

- クライアントアプリの業務概念を表すエンティティの定義
- ユーザー体験として利用する例外型の定義
- データの取得や保存などを含む業務ロジックの記述



```
dependencies:  
  freezed_annotation:  
  repository:  
    path: ../../repository  
  riverpod:  
  riverpod_annotation:  
  
dev_dependencies:  
  freezed:  
  riverpod_generator:
```

- Flutter への依存: transitive
- repository パッケージに依存してデータにアクセスする
- freezed で業務概念のエンティティを定義する
- riverpod で状態管理ロジックを実装する

```
@freezed  
class Member with _$Member {  
  const factory Member({  
    required int userId,  
    /** 省略 */  
    required LogInStatus loginStatus,  
  }) = _Member;  
  
  factory Member.fromDto(MemberDto dto) => Member(  
    userId: dto.userId,  
    /** 省略 */  
    loginStatus: LogInStatus.fromDateTime(dto.lastAccessedAt),  
  );  
}
```

- マッチングアプリにおけるお相手メンバーの業務概念の定義の例
- HTTP レスポンスに対応する Dto からエンティティを生成する

```
enum LogInStatus {  
    active,  
  
    recent,  
  
    inactive,  
}  
  
factory LogInStatus.fromDateTime(DateTime lastAccessedAt) {/** 省略 */}  
}
```

- お相手メンバーのログインステータス（直近どのくらいアクティブか）の定義
- 最終ログイン日時というサーバサイド(DB)上の事実から、ログインステータスというクライアントアプリにおける業務概念を計算する

```
@riverpod
Future<Member> memberDetail(MemberDetailRef ref, {required int userId}) async {
    final result = await ref.read(memberRepositoryProvider).fetchMember(userId);
    switch (result) {
        case SuccessRepositoryResult(:final data):
            return Member.fromDto(data);
        case FailureRepositoryResult(/** 省略 **/):
            /** 省略 */
    }
}
```

- riverpod の関数で取得処理のロジックを記述する
- repository による取得結果を switch 文で分岐して処理する

```
@Riverpod(keepAlive: true)
class AccountDetailNotifier extends _$AccountDetailNotifier {
    /** 省略 */

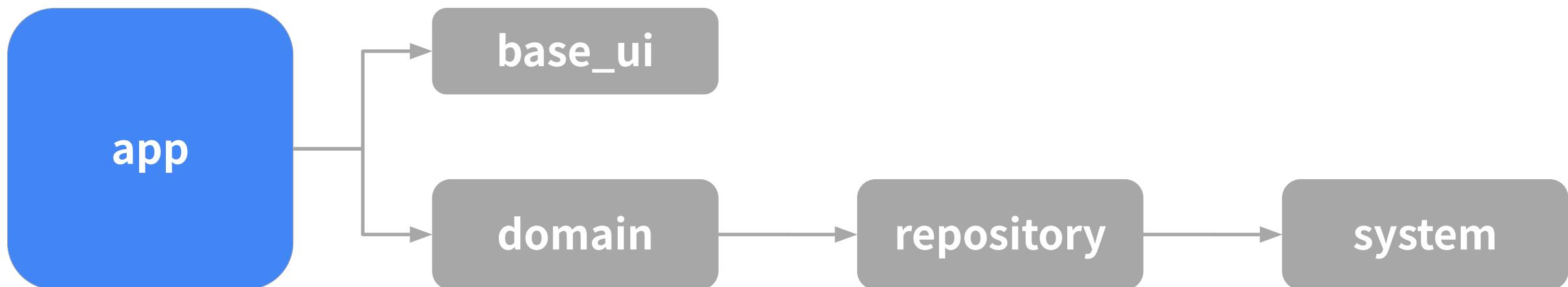
    Future<void> updateAccountDetail() async {
        final result = await ref.read(accountDetailRepositoryProvider).updateAccountDetail(** 省略 **);
        switch (result) {
            case SuccessRepositoryResult(** 省略 **):
                /** 省略 */
            case FailureRepositoryResult(** 省略 **):
                /** 省略 */
        }
    }
}
```

- riverpod のクラス (Notifier) で状態管理のロジックを記述する

```
class SendLikeUseCase {  
    /** 省略 */  
  
    Future<void> invoke/** 省略 */ async {  
        final result = await _ref.read(likeRepositoryProvider).sendLike/** 省略 */;  
        switch (result) {  
            /** 省略 */  
            case FailureRepositoryResult/** 省略 */:  
                /** 省略 */  
                throw InsufficientPointsToSendLikeException();  
        }  
    }  
}  
  
class InsufficientPointsToSendLikeException implements Exception {}
```

- お相手に「いいね！」を送信するロジック（状態管理を伴わない）
- 業務ロジックとして取り扱うべき例外を定義し、スローする

- Flutter アプリのエントリポイントとしての実装
 - `ProviderScope.overrides` による依存の注入
- UI 層としてのユーザー体験を実装
 - ウィジェットツリーの構築、画面遷移、各画面やコンポーネントの描画、ユーザー操作とのインタラクション



```
dependencies:  
  auto_route:  
  base_ui:  
    path: .. /base_ui  
  domain:  
    path: .. /domain  
  flutter:  
    sdk: flutter  
  flutter_hooks:  
  hooks_riverpod:  
  
dev_dependencies:  
  auto_route_generator:
```

- Flutter への依存: direct
- base_ui, domain パッケージに依存する
- auto_route, flutter_hooks など UI 層で利用するパッケージに依存する

```
@RoutePage()
class MemberProfilePage extends StatelessWidget {
  const MemberProfilePage({
    @PathParam('userId') required this.userId,
    super.key,
  });

  static String resolvePath({required String userId}) => '/users/$userId';

  final int userId;

  @override
  Widget build(BuildContext context) {/** 省略 */}
}
```

- auto_route パッケージを利用してルート定義

- domain 層の取得ロジックを `ref.watch` してユーザーが目にする体験を実装する

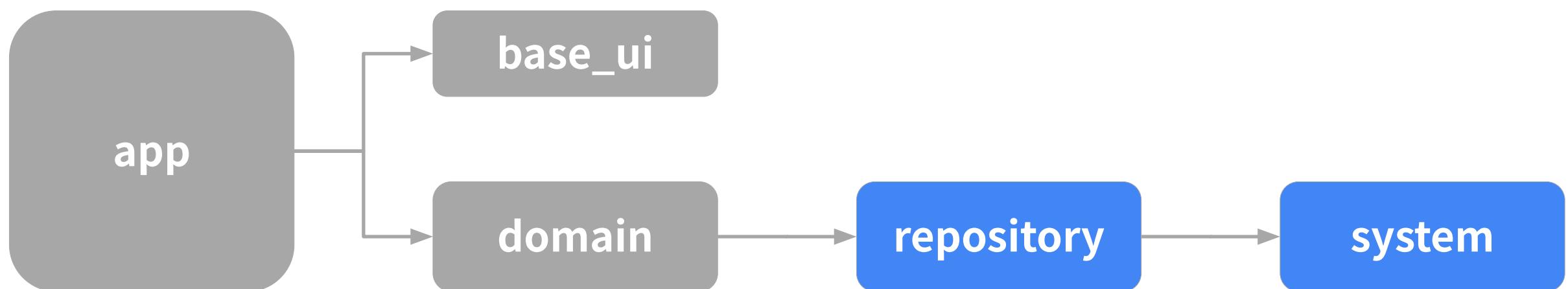
```
@override  
Widget build(BuildContext context, WidgetRef ref) {  
    // メンバー情報を取得する。  
    final memberAsyncValue = ref.watch(memberDetailProvider(userId: userId));  
    return switch (memberAsyncValue) {  
        AsyncData(value: final member) => /** 省略 **/,  
        AsyncError() => /** 省略 **/,  
        _ => /** 省略 **/,  
    };  
}
```

```
CommonFilledButton.sWithIcon(  
    onPressed: () async {  
        try {  
            await _ref.read(sendLikeUseCaseProvider).invoke(/** 省略 **/);  
        } on InsufficientPointsToSendLikeException catch (e) {  
            await _showErrorDialog(/** 省略 **/);  
        }  
        /** 省略 */  
    }  
    /** 省略 */  
)
```

- UI を通じたインタラクションをトリガーに domain 層のロジックを呼び出す
- domain 層に定義された例外を実現したいユーザーエクスペリエンスに従って処理する

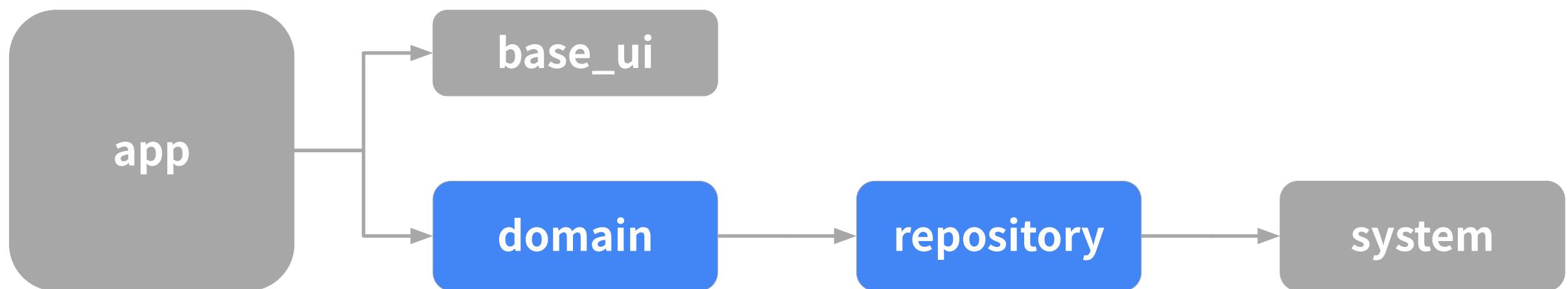
repository - system 間

- dio パッケージを通じて HTTP リクエストを行う
- dio パッケージへの依存は例外も含めて repository に表出しない
- Result 型のような `HttpResponse` 型をインターフェースとして、repository 層での `switch` による成功・失敗のハンドリングが強制、画一化される



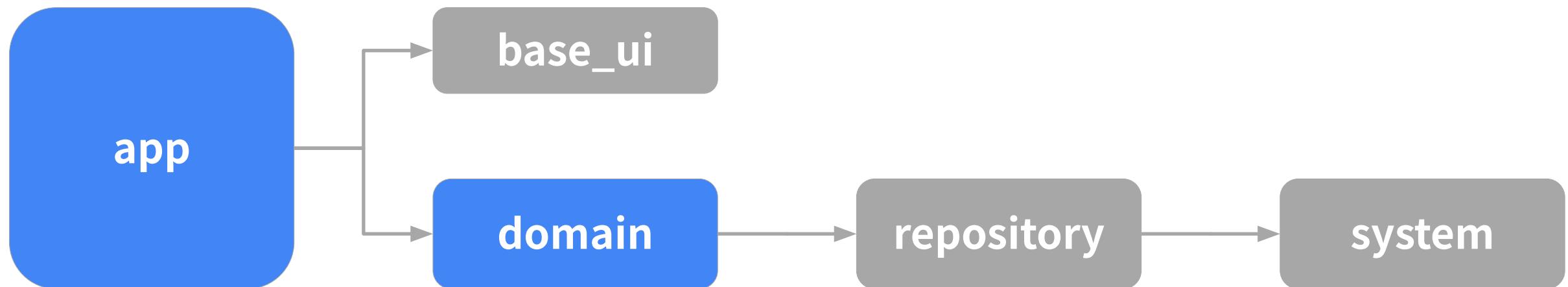
domain - repository 間

- repository のメソッドを呼び出してデータを読み書きする
- Result 型のような `RepositoryResult` 型をインターフェースとして、domain 層での `switch` による成功・失敗のハンドリングが強制、画一化される
- domain 層では Dto から業務知識を表すエンティティを生成し、サーバサイドの負債は持ち込まない



app - domain 間

- domain は業務概念を表すデータと業務ロジックを提供する
- app はユーザーエクスペリエンスの実現のために domain のロジックを利用し、業務概念のデータを読み書きする
- domain は業務知識として取り扱うべき例外をスローする
- app はそれを捕捉してユーザーエクスペリエンスに反映する



別パッケージにアクセストークンを取得するインターフェースのみを定義する：

```
@riverpod  
Future<String?> Function() extractAccessToken(Ref ref) => throw UnimplementedError();
```

インターフェースを通じてアクセストークンを取得して利用する：

```
class SomeRepository {  
  const SomeRepository(this._ref);  
  
  final Ref _ref;  
  
  Future<Something> fetchSomething() async {  
    final accessToken = await _ref.read(extractAccessTokenProvider);  
    /** 省略 */  
  }  
}
```

- アプリの動作時は app の `ProviderScope.overrides` で振る舞いを上書きする
- domain 層で保持・管理されているアクセストークンを必要なときに取り出す

```
ProviderScope.overrides(  
    overrides: [  
        extractAccessTokenProvider.overrideWith(  
            (ref) => () async => (await ref.read(authNotifierProvider.future))?.accessToken  
        ),  
    ],  
);
```

Tips : アクセストークンを通信層で利用する

Omiai

- 通信層のユニットテストでは、**ProviderContainer** で振る舞いを上書きする

```
ProviderContainer createContainer({  
    /** 省略 */  
    bool setAccessToken = true,  
}) {  
    final container = ProviderContainer(  
        overrides: [  
            accessTokenProvider.overrideWith(  
                (_) => () async => setAccessToken ? 'sample-access-token' : null,  
            ),  
            /** 省略 */  
        ],  
            /** 省略 */  
    );  
    addTearDown(container.dispose);  
    return container;  
}
```

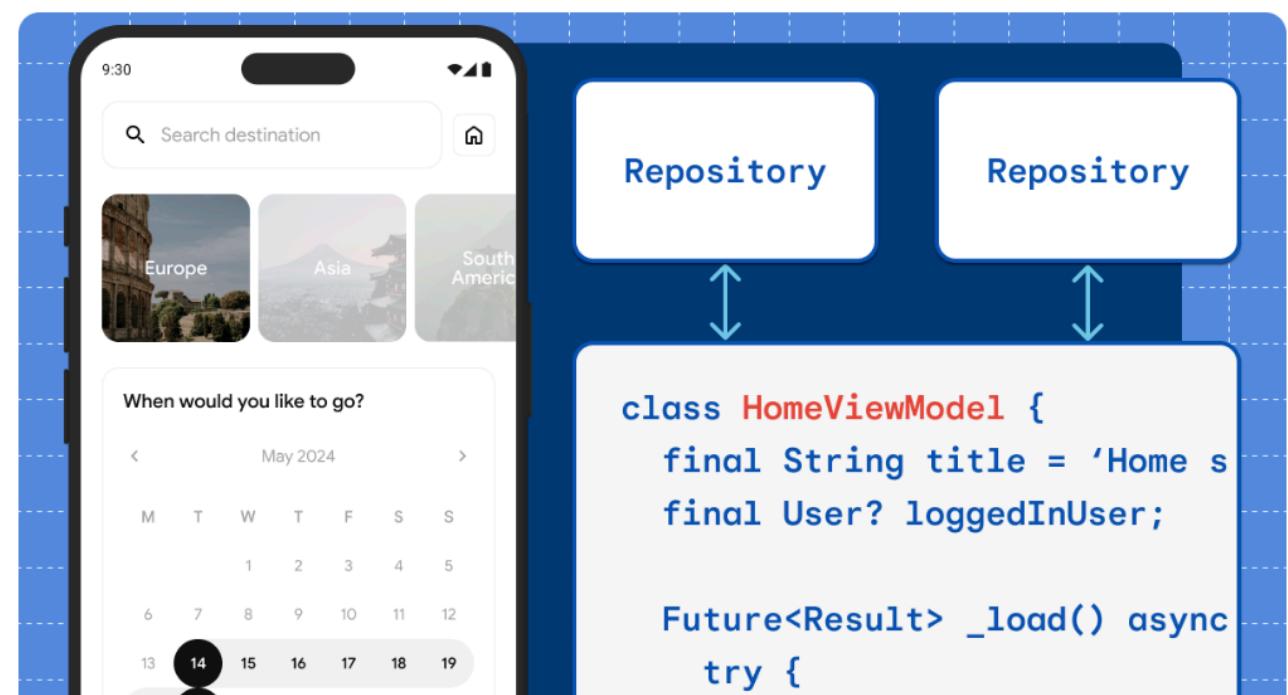
[Architecture concepts >](#)

Architecting Flutter apps

Architecture

Architecture is an important part of building a maintainable, resilient, and scalable Flutter app. In this guide, you'll learn app architecture principles and best practices for building Flutter apps.

'Architecture' is a word that's hard to define. It's a broad term and can refer to any number of topics depending on the context. In this guide, 'architecture' refers to how to structure, organize, and design your Flutter app in order to scale as your project requirements and team



🤔 build_runner を各パッケージで実行するのが面倒

- 仕方なし。エイリアスやタスクランナーなどの工夫を！

🤔 同じ機能に関する実装が Explorer 上で遠く見える

- ファイル検索の仕方を工夫すると Explorer 上でファイルを探すことがなくなる

🤔 冗長なコードが増える可能性がある

- サーバサイドの概念、命名、テーブル定義、HTTP インターフェース、クライアントアプリで取り扱う概念に差がない場合など

🤔 難しそう・うまく運用できるか不安

- CI や Melos へのキャッチアップは一定必要だが、単一パッケージの構成に戻すのは簡単！

ここまで

パッケージを分けることで...

- 依存に関する誤った実装を仕組み上防ぐことができる
- 各レイヤーの責務が明確になり、ユニットテストも容易に書ける
- Result 相当のインターフェース定義と合わせて、レイヤーを跨ぐ例外ハンドリングがパターン化される
- 「いま自分は Flutter エンジニアなのか、Dart エンジニアなのか」の意識を明確にして開発できる
- クライアントアプリとサーバサイドアプリとの境界が明確になり、サーバサイドの負債をクライアントアプリに持ち込まなくなる

リプレイスの成功のための更なる取り組み

✓ ユニットテスト

- domain および、それより抽象的なレイヤーでテストカバレッジ 100%
- CI で継続的にカバレッジを計測

✓ ドキュメンテーション

- public_member_api_docs を全面的に有効化して doc comment を 100% 記述

✓ 実装の可能な限りのパターン化

- マッチングアプリはそんなに難しくない！
- 生成 AI を活用する試み

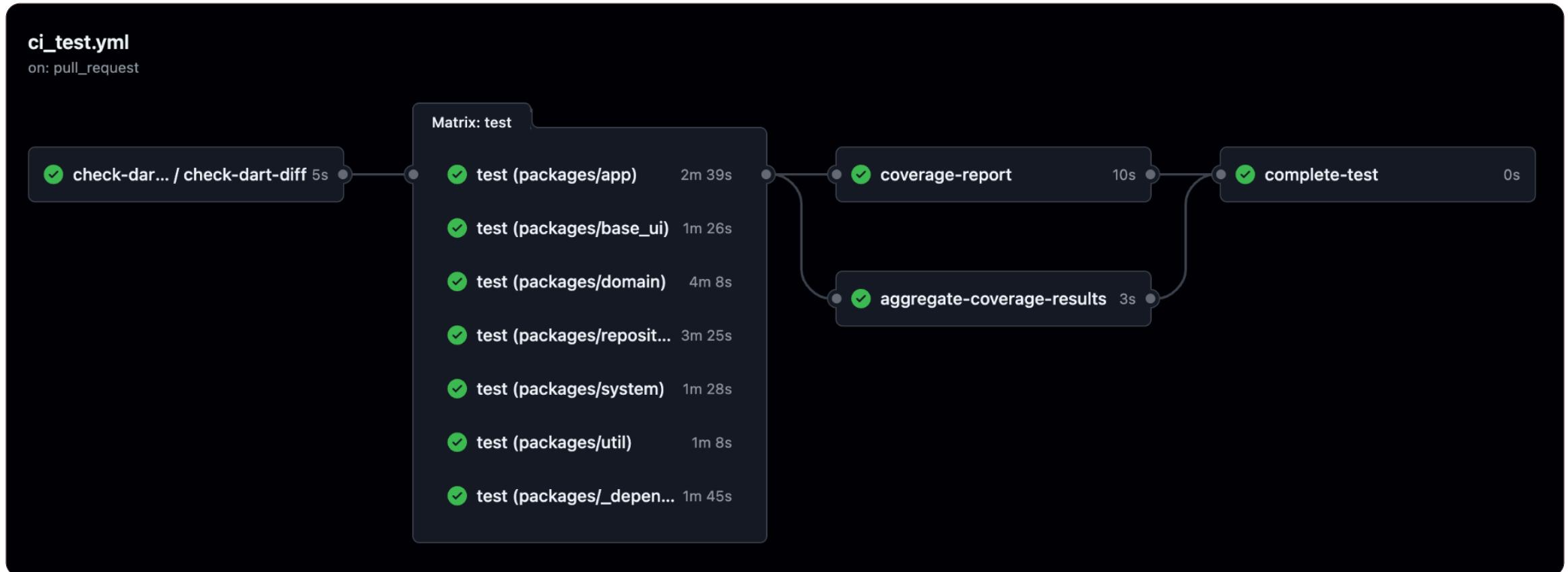
ユニットテスト

Test Scope と Test Size

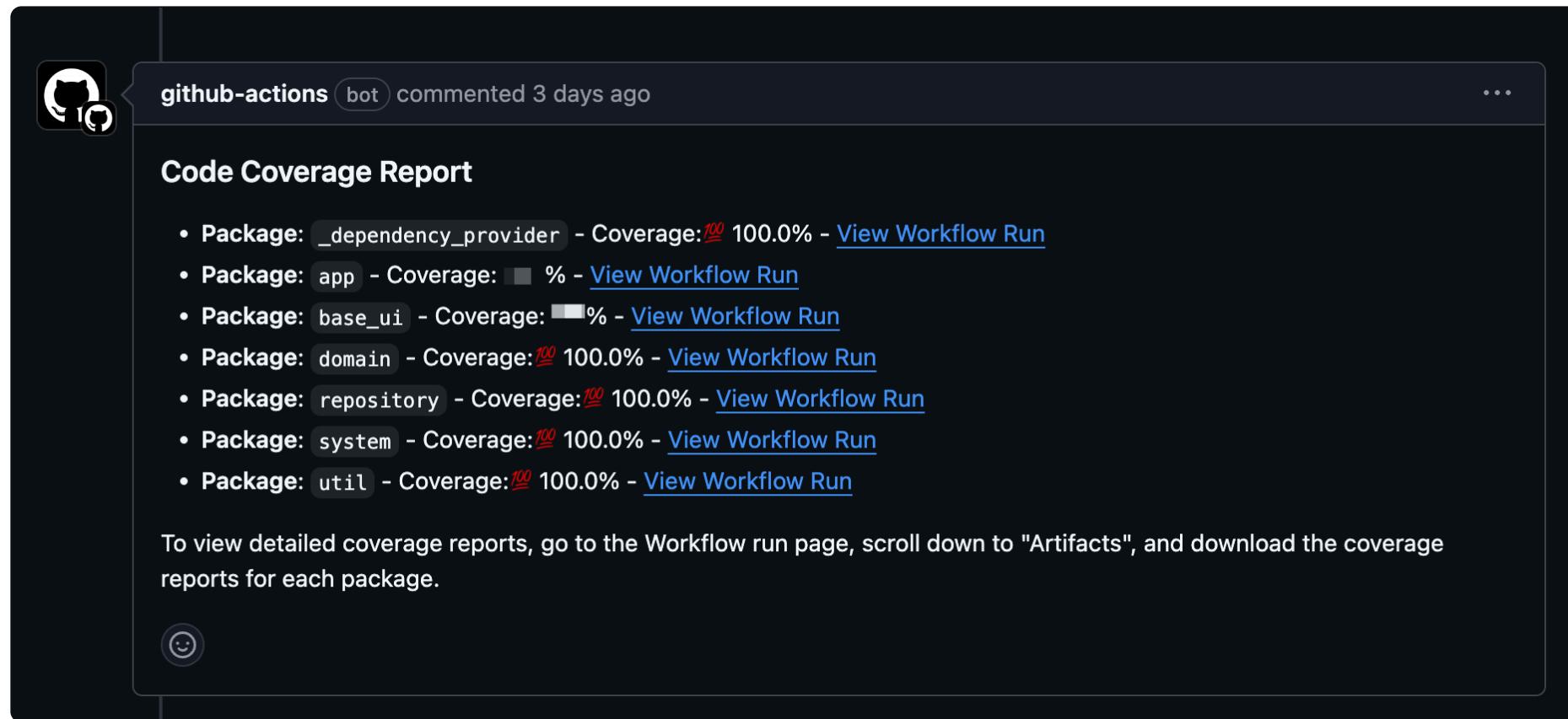
		Test Size		
		Small (単一プロセス)	Medium (単一マシン)	Large (制約なし)
Test Scope	Unit (比率 80%)	大いに推奨	避けたいが しかたないときも	最悪だが よく見かける
	Integration (比率 15%)	書けるなら コスパ良し	普通	できれば 避けたい
	E2E (比率 5%)	原理上不可能に近いが 小さいシステムなら可能？	小さいシステム なら可能	普通かつ CUJに絞りたい

出典：「自動テストの種類の曖昧さが少ない「テストサイズ」という分類」（ログミーTech）

- matrix を組んで各パッケージのユニットテストを CI で継続的に実行する



- 業務ロジック層およびそれより抽象的な層でのユニットテストのカバレッジが100%であることをPR上で確認する



ドキュメンテーションコメント

public_member_api_docs

[Tools](#) > [Linter rules](#) > public_member_api_docs

Document all public members.

This rule is available as of Dart 2.0.

Details

DO document all public members.

All non-overriding public members should be documented with `///` doc-style comments.

BAD:

```
class Bad {  
    void meh() {}  
}
```

dart

ドキュメンテーションコメント

Omiai

```
/// firebase_analytics パッケージが提供する、Firebase Analytics の各機能をラップしたクラス。  
///  
/// ほとんど firebase_analytics のインターフェースの再実装のような内容になっているが、  
///  
/// - system パッケージ以外が firebase_analytics パッケージに直接依存しないようにする  
/// - 分析ログの送信という性質上、利用する側に await させず、例外ハンドリングもさせない（例外が  
/// 起きても呼び出し側の処理を止めない）ようにする  
/// - 一種の腐敗防止層としての実装とする  
///  
/// ことなどを目的としている。  
class FirebaseAnalyticsClient {  
    /// [FirebaseAnalyticsClient] を生成する。  
    const FirebaseAnalyticsClient(this._firebaseAnalytics);  
  
    final FirebaseAnalytics _firebaseAnalytics;  
  
    /// Firebase Analytics にログを送信する。  
    ///  
    /// 本メソッドの返り値型は `void` で定義されており、[unawaited] で [_logEvent] を呼び  
    /// 出すことでの、もし例外が発生しても呼び出し側の処理を止めない（ただし、グローバルエラー  
    /// ハンドラーを定義すれば、キャッチできる）ようにしている。  
    void logEvent({required String name, required Map<String, Object>? parameters}) =>  
        unawaited(_logEvent(name: name, parameters: parameters));  
  
    /** 省略 */  
}
```

- public_member_api_docs の lint ルールを全面的に有効化
- パッケージ分割と合わせて、「パッケージを作るよう アプリケーションを作る」意識を重視
- インターフェースが完璧で、ドキュメンテーションが十分で、ユニットテストが十分に書かれた実装に依存する安心・優れた開発体験
- ソースコードが「実行可能な仕様書」としての理想に近づく

実装の可能な限りのパターン化

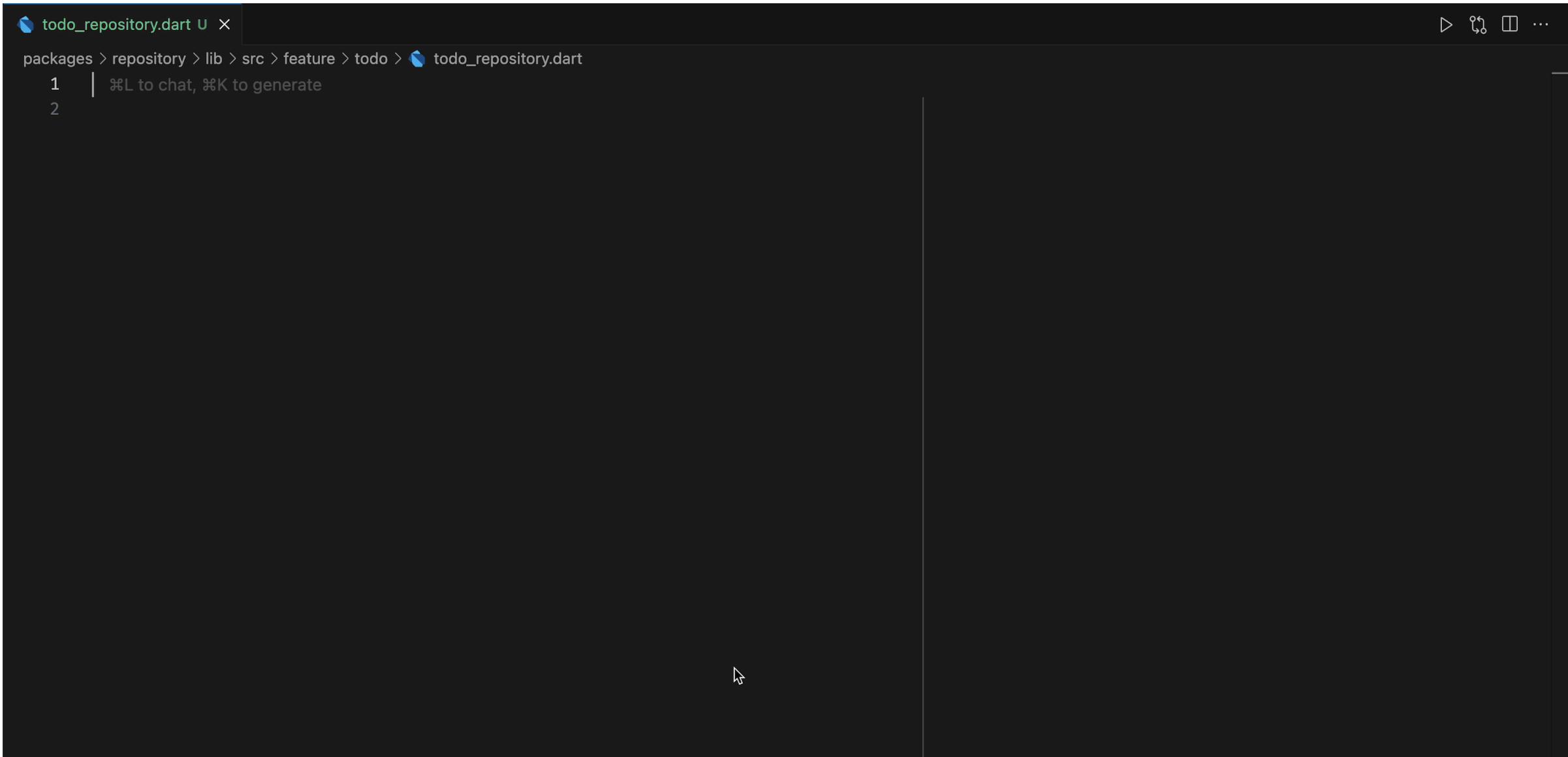
✓ スニペットの充実

- Omiai では合計 20 個のスニペットを定義
- 各レイヤーで必要になる典型的な実装はすべてスニペット化済み

✓ 堅牢な設計がパターン化を支える

- 例外ハンドリングもスニペット化
- ユニットテストもスニペット化
- doc comment もスニペットに含まれている
- 開発者は、インターフェース設計やるべき業務概念を考えること、UI 層の作り込みに集中できる
- スニペットでカバーできない実装が見つかれば、不確実性の高いタスクとして認識可能

例：Repository の実装



A screenshot of a code editor window titled "todo_repository.dart". The window shows a Dart file structure:

```
packages > repository > lib > src > feature > todo > todo_repository.dart
```

The code editor interface includes a status bar at the top with icons for file operations and a navigation bar with icons for search, refresh, and more.

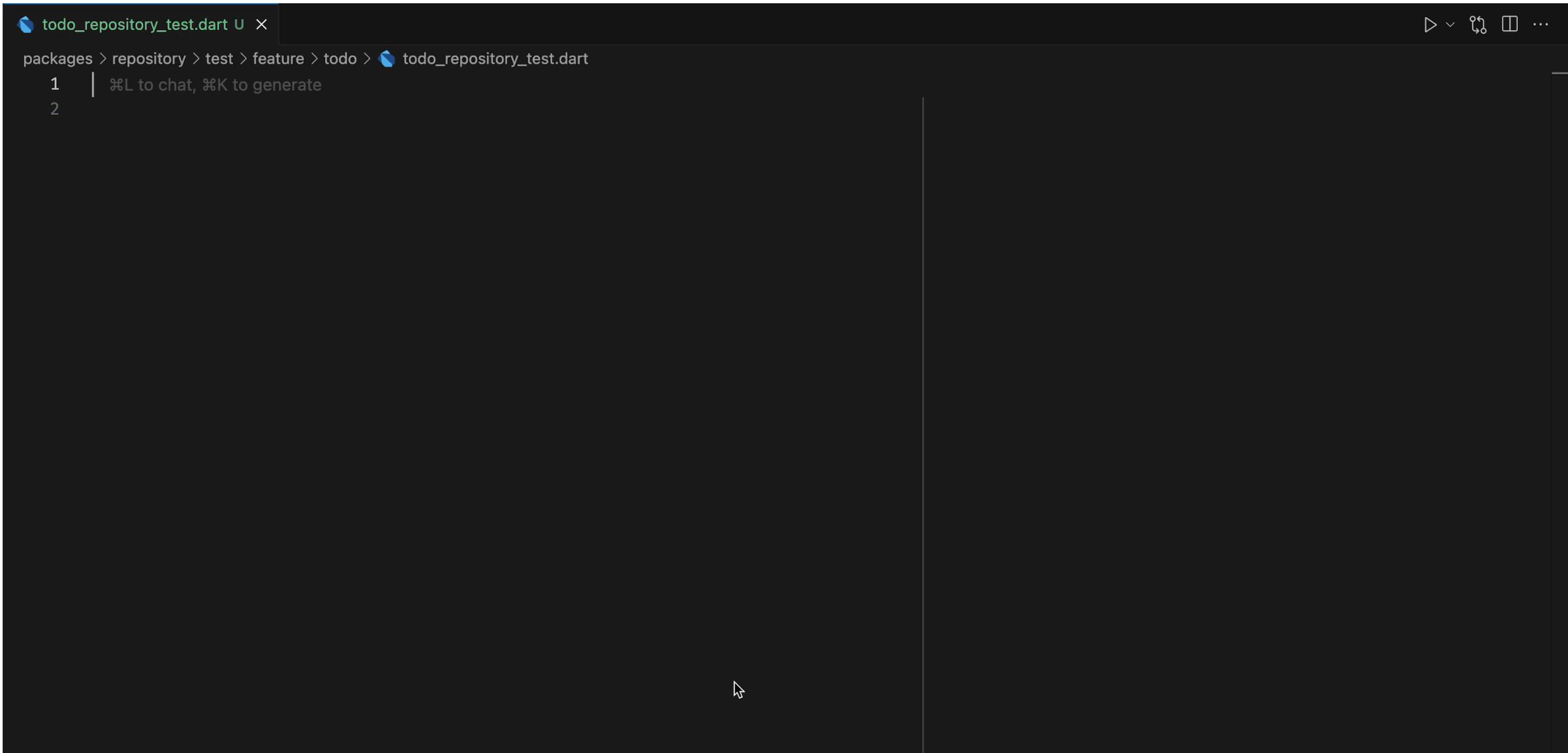
The code itself consists of two lines of code:

```
1 | #L to chat, #K to generate
2
```

The cursor is positioned at the end of the second line.

例：Repository のテスト

Omi**ai**



A screenshot of a dark-themed code editor window. The title bar shows the file name "todo_repository_test.dart". The status bar at the bottom indicates the file path: "packages > repository > test > feature > todo > todo_repository_test.dart". The code editor displays two lines of code:

```
1 | // L to chat, K to generate
2
```

The editor has a light gray background with dark gray syntax highlighting. The status bar includes icons for navigation and file operations.

試み：生成 AI の活用

Omi**ai**

A screenshot of a dark-themed code editor interface, likely from a tool like Omiai. The top navigation bar includes tabs for 'todo_repository.dart' (marked with a blue icon), 'todo_dto.dart' (marked with a blue icon), and 'todo_repository_test.dart' (marked with a blue icon). Below the tabs, a breadcrumb path shows the file structure: 'packages > repository > test > feature > todo > todo_repository_test.dart'. The main workspace contains two lines of code: '1 | %L to chat, %K to generate' and '2'. To the right of the workspace is a vertical sidebar containing three tabs: 'CHAT', 'COMPOSER' (which is currently selected and highlighted in blue), and 'REVIEW'. The 'COMPOSER' tab features a text input field with placeholder text '+ Add context' and instructions 'Edit, refactor, or add code (/ for commands)'. A small note at the bottom of the input field says 'Esc to blur'. At the bottom right of the sidebar, there is a dropdown menu labeled 'claude-3.5-sonnet' and a 'Submit' button. A cursor arrow is visible at the bottom center of the screen.

```
part '${TM_FILENAME_BASE}.g.dart';

class ${TM_FILENAME_BASE/(.*)_use_case${1:/pascalcase}/}UseCase {
  /** 省略 */
}
```

- `TM_FILENAME_BASE` や `TM_DIRECTORY` のような Variables が利用可能

まとめ

- 大規模アプリのリプレイスの成功のための堅牢な設計
 - Architecting Flutter apps や Riverpod の公式ドキュメントをベースに
 - 更にパッケージ分割で負債を仕組みで未然に防ぐ
 - 定めた対象に対してテストを十分に記述し、継続的にカバレッジなどを把握する
 - ドキュメンテーションコメントを必須にして「実行可能な仕様書」の理想へ
- 堅牢な設計の副次的な恩恵
 - 各レイヤーの責務や依存が明確で、それぞれの実装が単純化・パターン化される
 - スニペットの充実は開発体験の向上と品質の維持に有効で、生成 AI の活用も期待される