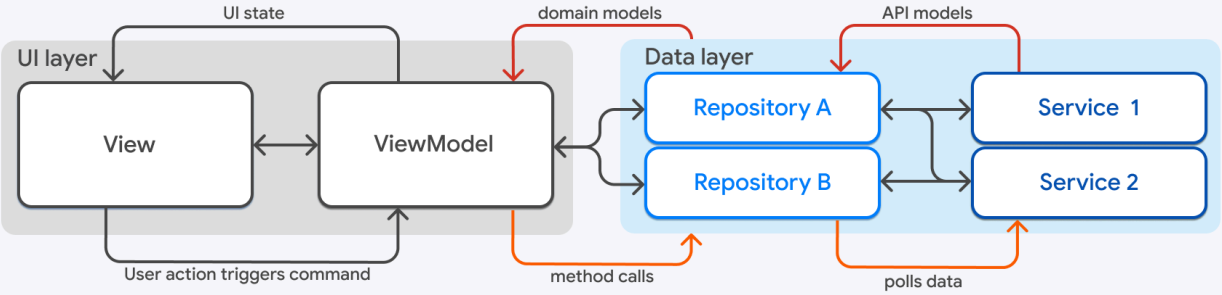
Data layer

Contents

The data layer of an application, known as the *model* in MVVM terminology, is the source of truth for all application data. As the source of truth, it's the only place that application data should be updated.

It's responsible for consuming data from various external APIs, exposing that data to the UI, handling events from the UI that require data to be updated, and sending update requests to those external APIs as needed.

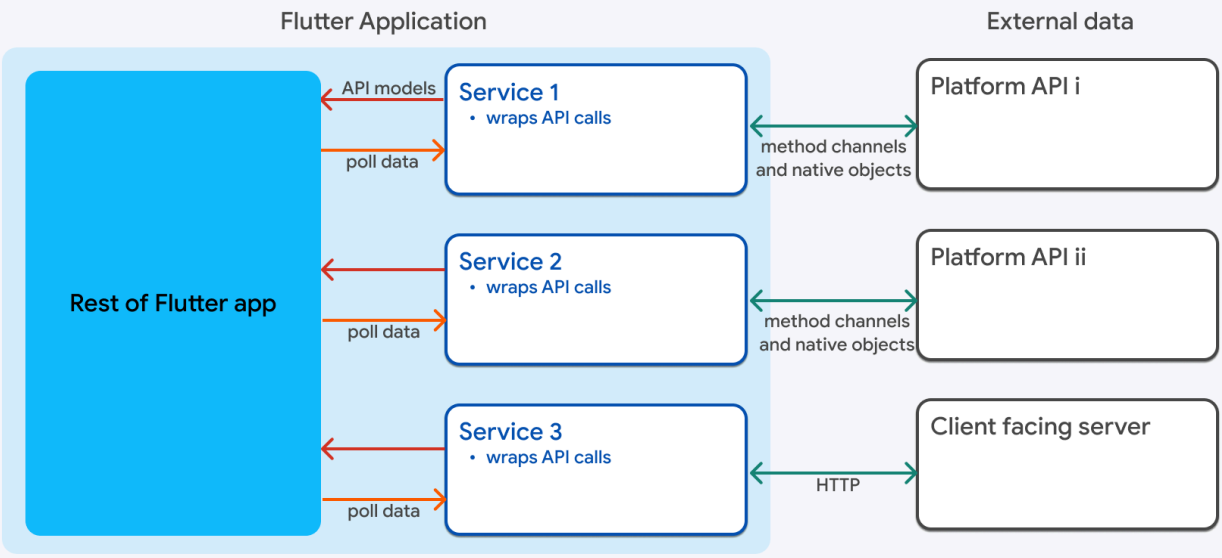
The data layer in this guide has two main components, repositories and services.



- **Repositories** are the source of the truth for application data, and contain logic that relates to that data, like updating the data in response to new user events or polling for data from services. Repositories are responsible for synchronizing the data when offline capabilities are supported, managing retry logic, and caching data.
- **Services** are stateless Dart classes that interact with APIs, like HTTP servers and platform plugins. Any data that your application needs that isn't created inside the application code itself should be fetched from within service classes.

Define a service

A service class is the least ambiguous of all the architecture components. It's stateless, and its functions don't have side effects. Its only job is to wrap an external API. There's generally one service class per data source, such as a client HTTP server or a platform plugin.



In the Compass app, for example, there's an APIClient service that handles the CRUD calls to the client-facing server.

api_client.dart

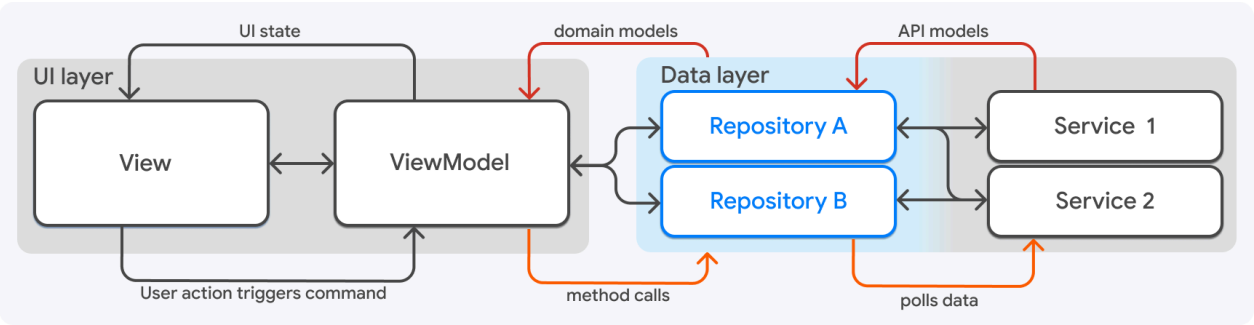
```dart
class ApiClient {
  // Some code omitted for demo purposes.

  Future<Result<List<ContinentApiModel>>> getContinents() async { /* ... */ }

  Future<Result<List<DestinationApiModel>>> getDestinations() async { /* ... */ }

  Future<Result<List<ActivityApiModel>>> getActivityByDestination(String ref) async { /* ... */ }

  Future<Result<List<BookingApiModel>>> getBookings() async { /* ... */ }

  Future<Result<BookingApiModel>> getBooking(int id) async { /* ... */ }

  Future<Result<BookingApiModel>> postBooking(BookingApiModel booking) async { /* ... */ }

  Future<Result<void>> deleteBooking(int id) async { /* ... */ }

  Future<Result<UserApiModel>> getUser() async { /* ... */ }
}
```

The service itself is a class, where each method wraps a different API endpoint and exposes asynchronous response objects. Continuing the earlier example of deleting a saved booking, the `deleteBooking` method returns a `Future<Result<void>>`.

> **Note**
>
> Some methods return data classes that are specifically for raw data from the API, such as the `BookingApiModel` class. As you'll soon see, repositories extract data and expose it to s in a different format.

Define a repository

A repository's sole responsibility is to manage application data. A repository is the source of truth for a single type of application data, and it should be the only place where that data type is mutated. The repository is responsible for polling new data from external sources, handling retry logic, managing cached data, and transforming raw data into domain models.



You should have a separate repository for each different type of data in your application. For example, the Compass app has repositories called `UserRepository`, `BookingRepository`, `AuthRepository`, `DestinationRepository`, and more.

The following example is the `BookingRepository` from the Compass app, and shows the basic structure of a repository.

booking_repository_remote.dart

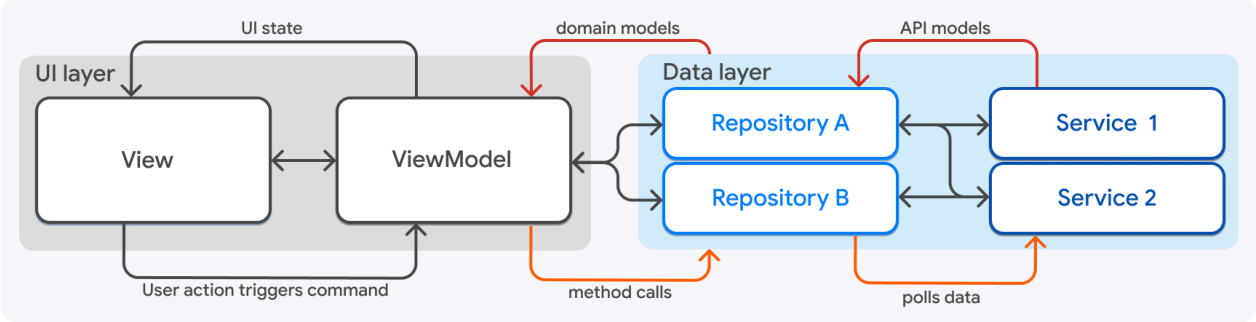
```dart
class BookingRepositoryRemote implements BookingRepository {
  BookingRepositoryRemote({
    required ApiClient apiClient,
  }) : _apiClient = apiClient;

  final ApiClient _apiClient;
  List<Destination>? _cachedDestinations;

  Future<Result<void>> createBooking(Booking booking) async {...}
  Future<Result<Booking>> getBooking(int id) async {...}
  Future<Result<List<BookingSummary>>> getBookingsList() async {...}
  Future<Result<void>> delete(int id) async {...}
}
```

> **Development versus staging environments**
>
> The class in the previous example is `BookingRepositoryRemote`, which extends an abstract class called `BookingRepository`. This base class is used to create repositories for different environments. For example, the compass app also has a class called `BookingRepositoryLocal`, which is used for local development.
>
> You can see the differences between the `BookingRepository` classes on GitHub.

The `BookingRepository` takes the `ApiClient` service as an input, which it uses to get and update the raw data from the server. It's important that the service is a private member, so that the UI layer can't bypass the repository and call a service directly.

With the `ApiClient` service, the repository can poll for updates to a user's saved bookings that might happen on the server, and make `POST` requests to delete saved bookings.

The raw data that a repository transforms into application models can come from multiple sources and multiple services, and therefore repositories and services have a many-to-many relationship. A service can be used by any number of repositories, and a repository can use more than one service.



Domain models

The `BookingRepository` outputs `Booking` and `BookingSummary` objects, which are *domain models*. All repositories output corresponding domain models. These data models differ from API models in that they only contain the data needed by the rest of the app. API models contain raw data that often needs to be filtered, combined, or deleted to be useful to the app's s. The repo refines the raw data and outputs it as domain models.

In the example app, domain models are exposed through return values on methods like `BookingRepository.getBooking`. The `getBooking` method is responsible for getting the raw data from the `ApiClient` service, and transforming it into a `Booking` object. It does this by combining data from multiple service endpoints.

booking_repository_remote.dart

```dart
// This method was edited for brevity.
Future<Result<Booking>> getBooking(int id) async {
  try {
    // Get the booking by ID from server.
    final resultBooking = await _apiClient.getBooking(id);
    if (resultBooking is Error<BookingApiModel>) {
      return Result.error(resultBooking.error);
    }
    final booking = resultBooking.asOk.value;

    final destination = _apiClient.getDestination(booking.destinationRef);
    final activities = _apiClient.getActivitiesForBooking(
            booking.activitiesRef);

    return Result.ok(
      Booking(
        startDate: booking.startDate,
        endDate: booking.endDate,
        destination: destination,
        activity: activities,
      ),
    );
  } on Exception catch (e) {
    return Result.error(e);
  }
}
```

> **Note**
>
> In the Compass app, service classes return `Result` objects. `Result` is a utility class that wraps asynchronous calls and makes it easier to handle errors and manage UI state that relies on asynchronous calls.
>
> This pattern is a recommendation, but not a requirement. The architecture recommended in this guide can be implemented without it.
>
> You can learn about this class in the Result cookbook recipe.

Complete the event cycle

Throughout this page, you've seen how a user can delete a saved booking, starting with an event—a user swiping on a `Dismissible` widget. The view model handles that event by delegating the actual data mutation to the `BookingRepository`. The following snippet shows the `BookingRepository.deleteBooking` method.

booking_repository_remote.dart

```dart
Future<Result<void>> delete(int id) async {
  try {
    return _apiClient.deleteBooking(id);
  } on Exception catch (e) {
    return Result.error(e);
  }
}
```

The repository sends a `POST` request to the API client with the `_apiClient.deleteBooking` method, and returns a `Result`. The `HomeViewModel` consumes the `Result, and the data it contains, and ultimately calls`notifyListeners`, completing the cycle.

Feedback

As this section of the website is evolving, we welcome your feedback!

〈 UI layer

Dependency Injection 〉