Testing each layer

Contents

Testing the UI layer
  ViewModel unit tests
  View widget tests
Testing the data layer
Feedback

Testing the UI layer

One way to determine whether your architecture is sound is considering how easy (or difficult) the application is to test. Because view models and views have well-defined inputs, their dependencies can easily be mocked or faked, and unit tests are easily written.

ViewModel unit tests

To test the UI logic of the view model, you should write unit tests that don't rely on Flutter libraries or testing frameworks.

Repositories are a view model's only dependencies (unless you're implementing use-cases), and writing `mocks` or `fakes` of the repository is the only setup you need to do. In this example test, a fake called `FakeBookingRepository` is used.

home_screen_test.dart

```dart
void main() {
  group('HomeViewModel tests', () {
    test('Load bookings', () {
      // HomeViewModel._load is called in the constructor of HomeViewModel.
      final viewModel = HomeViewModel(
        bookingRepository: FakeBookingRepository()
          ..createBooking(kBooking),
        userRepository: FakeUserRepository(),
      );

      expect(viewModel.bookings.isNotEmpty, true);
    });
  });
}
```

The `FakeBookingRepository` class implements `BookingRepository`. In the data layer section of this case-study, the `BookingRepository` class is explained thoroughly.

fake_booking_repository.dart

```dart
class FakeBookingRepository implements BookingRepository {
  List<Booking> bookings = List.empty(growable: true);

  @override
  Future<Result<void>> createBooking(Booking booking) async {
    bookings.add(booking);
    return Result.ok(null);
  }
  // ...
}
```

> **Note**
>
> If you're using this architecture with use-cases, these would similarly need to be faked.

View widget tests

Once you've written tests for your view model, you've already created the fakes you need to write widget tests as well. The following example shows how the `HomeScreen` widget tests are set up using the `HomeViewModel` and needed repositories:

home_screen_test.dart

```dart
void main() {
  group('HomeScreen tests', () {
    late HomeViewModel viewModel;
    late MockGoRouter goRouter;
    late FakeBookingRepository bookingRepository;

    setUp(() {
      bookingRepository = FakeBookingRepository()
        ..createBooking(kBooking);
      viewModel = HomeViewModel(
        bookingRepository: bookingRepository,
        userRepository: FakeUserRepository(),
      );
      goRouter = MockGoRouter();
      when(() => goRouter.push(any())).thenAnswer((_) => Future.value(null));
    });

    // ...
  });
}
```

This setup creates the two fake repositories needed, and passes them into a `HomeViewModel` object. This class doesn't need to be faked.

> **Note**
>
> The code also defines a `MockGoRouter`. The router is mocked using `package:mocktail`, and is outside the scope of this case-study. You can find general testing guidance in Flutter's testing documentation.

After the view model and its dependencies are defined, the Widget tree that will be tested needs to be created. In the tests for `HomeScreen`, a `loadWidget` method is defined.

home_screen_test.dart

```dart
void main() {
  group('HomeScreen tests', () {
    late HomeViewModel viewModel;
    late MockGoRouter goRouter;
    late FakeBookingRepository bookingRepository;

    setUp(
      // ...
    );

    void loadWidget(WidgetTester tester) async {
      await testApp(
        tester,
        ChangeNotifierProvider.value(
          value: FakeAuthRepository() as AuthRepository,
          child: Provider.value(
            value: FakeItineraryConfigRepository() as ItineraryConfigRepository,
            child: HomeScreen(viewModel: viewModel),
          ),
        ),
        goRouter: goRouter,
      );
    }

    // ...
  });
}
```

This method turns around and calls `testApp`, a generalized method used for all widget tests in the compass app. It looks like this:

testing/app.dart

```dart
void testApp(
  WidgetTester tester,
  Widget body, {
  GoRouter? goRouter,
}) async {
  tester.view.devicePixelRatio = 1.0;
  await tester.binding.setSurfaceSize(const Size(1200, 800));
  await mockNetworkImages(() async {
    await tester.pumpWidget(
      MaterialApp(
        localizationsDelegates: [
          GlobalWidgetsLocalizations.delegate,
          GlobalMaterialLocalizations.delegate,
          AppLocalizationDelegate(),
        ],
        theme: AppTheme.lightTheme,
        home: InheritedGoRouter(
          goRouter: goRouter ?? MockGoRouter(),
          child: Scaffold(
            body: body,
          ),
        ),
      ),
    );
  });
}
```

This function's only job is to create a widget tree that can be tested.

The `loadWidget` method passes in the unique parts of a widget tree for testing. In this case, that includes the `HomeScreen` and its view model, as well as some additional faked repositories that are higher in the widget tree.

The most important thing to take away is that view and view model tests only require mocking repositories if your architecture is sound.

Testing the data layer

Similar to the UI layer, the components of the data layer have well-defined inputs and outputs, making both sides fake-able. To write unit tests for any given repository, mock the services that it depends on. The following example shows a unit test for the `BookingRepository`.

booking_repository_remote_test.dart

```dart
void main() {
  group('BookingRepositoryRemote tests', () {
    late BookingRepository bookingRepository;
    late FakeApiClient fakeApiClient;

    setUp(() {
      fakeApiClient = FakeApiClient();
      bookingRepository = BookingRepositoryRemote(
        apiClient: fakeApiClient,
      );
    });

    test('should get booking', () async {
      final result = await bookingRepository.getBooking(0);
      final booking = result.asOk.value;
      expect(booking, kBooking);
    });
  });
}
```

To learn more about writing mocks and fakes, check out examples in the Compass App `testing` directory or read Flutter's testing documentation.

Feedback

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

〈 Dependency injection