Testing Flutter App

Автоматическое тестирование делится на несколько категорий:

Unit test - тестирует одну функцию, метод или класс.

Widget test (в других фреймворках пользовательского интерфейса называемый тестом компонентов) - тестирует один виджет.

Integration test - тестирует все приложение или большую часть приложения.

Unit test

Модульный тест тестирует одну функцию, метод или класс. Цель модульного теста - проверить правильность логической единицы в различных условиях. Внешние зависимости тестируемого модуля обычно имитируются.

Dependency

dev_dependencies:

test: <latest_version>

Folder structure:

counter_app/

lib/

counter.dart

test/

counter_test.dart

Sample class for test

```
class Counter {
  int value = 0;

  void increment() => value++;

  void decrement() => value--;
}
```

Test

```
test ('Название теста', () {
  // Код теста
 });
```

```
import 'package:test/test.dart';
import 'package:counter_app/counter.dart';

void main() {
  test('Counter value should be incremented', () {
    final counter = Counter();
    counter.increment();
    expect(counter.value, 1);
  });
}
```

Combine tests

```
void main() {
  group('Counter', () {
    test('value should start at 0', () {
      expect(Counter().value, 0);
    });
    test('value should be incremented', () {
      final counter = Counter();
      counter.increment();
     expect(counter.value, 1);
    });
    test('value should be decremented', () {
      final counter = Counter();
      counter.decrement();
      expect(counter.value, -1);
    });
  });
}
```

Run the tests

IntelliJ/Android Studio:

Open the counter_test.dart file

Select the Run menu

Click the Run 'tests in counter_test.dart' option

VSCode:

Open the counter_test.dart file

Select the Run menu

Click the Start Debugging option

Terminal:

flutter test test/counter_test.dart

Widger testing

Целью теста является доказательство того, что пользовательский интерфейс виджета выглядит и взаимодействует так, как это запланировано. Тестирование виджета требует тестовую среду, которая обеспечивает соответствующий контекст жизненного цикла виджета.

Создание теста

```
testWidgets('Название теста', (WidgetTester tester) async {
    // Код теста
  });
```

Также можно группировать

```
group('Название группы тестов', (){
    testWidgets('Название теста', (WidgetTester tester) async {
        // Код теста
      });
    testWidgets('Название теста', (WidgetTester tester) async {
        // Код теста
      });
  });
```

Дополнительные настройки

```
setUp(() {
    // код инициализации теста
  });
tearDown(() {
    // код финализации теста
  });
setUpAll(() {
    // код инициализации всех тестов
  });
tearDownAll(() {
    // код финализации всех тестов
  });
```

Поиск виджетов

в дереве по тексту — **find.text**, **find.widgetWithText**;

по ключу — **find.byKey**;

по иконке — **find.byIcon**, **find.widgetWithIcon**;

по типу — **find.byType**;

по положению в дереве — **find.descendant** и **find.ancestor**;

глобальный объект find, который позволяет найти виджеты

Взаимодействие с тестируемым виджетом

Класс WidgetTester предоставляет функции для создания тестируемого виджета, ожидания смены его состояний и для выполнения некоторых действий над этими виджетами.
Любое изменение виджета вызывает изменение его состояния. Но тестовая среда не перестраивает виджет при этом.

pumpWidget — создание тестируемого виджета;

pump — запускает обработку смены состояния виджета и ожидает ее завершения в течении заданного таймаута (100 мс по умолчанию);

pumpAndSettle — вызывает pump в цикле для смены состояний в течении заданного таймаута (100 мс по умолчанию), это ожидание завершения всех анимаций;

tap — отправить виджету нажатие;

longPress — длинное нажатие;

fling — смахивание/свайп;

drag — перенос;

enterText — ввод текста.

После любых действий с виджетами нужно
вызывать **tester.pumpAndSettle()** для смены состояний.

Пример

```
await tester.enterText(find.byKey(Key('phoneField')), 'bla-bla-bla');
```

Проверки

В процессе выполнения теста можно проверить наличие виджетов на экране. Это позволяет убедиться в том, что новое состояние экрана корректно с точки зрения видимости нужных виджетов:

expect(find.text('Номер телефона'), findsOneWidget);

expect(find.text('Код из СМС'), findsNothing);

Mocks

Эта библиотека позволяет создать моковые классы, от которых зависит тестируемый виджет, для того, чтобы тест был более простым и охватывал только тот код, который мы тестируем.

```
dependencies:
  mockito: any
```

Пример

PhoneInputScreen, который при нажатии, используя **AuthInteractor** сервис, выполняет запрос к бэкенду **authInteractor.checkAccess()**, то подставив мок вместо сервиса, мы сможем проверить самое главное — наличие факта обращения к этому сервису.

class AuthInteractorMock extends Mock implements AuthInteractor {}

Для определения функциональности мока используется функция **when**, которая позволяет определить ответ мока на вызов той или иной функции:

```
when(
  authInteractor.checkAccess(any),
).thenAnswer((_) => Future.value(true));

when(
  authInteractor.checkAccess(any),
).thenAnswer((_) => Future.error(UnknownHttpStatusCode(null)));
```

После выполнения теста, также можно проверить какие методы мокового класса вызывались в ходе теста, и сколько раз. Это необходимо, например, для понимания того, не слишком ли часто запрашиваются те или иные данные, нет ли лишних изменений состояния приложения:

```
verify(appComponent.authInteractor).called(1);
verify(authInteractor.checkAccess(any)).called(1);
verifyNever(appComponent.profileInteractor);
```

Integration tests

В отличии от виджет-тестов, интеграционный тест проверяет полностью все приложение или какую-то большую его часть. Цель интеграционного теста заключается в том, чтобы убедиться, что все виджеты и сервисы работают вместе так, как и ожидалось. Процесс работы интеграционного теста можно наблюдать в симуляторе или на экране устройства. Этот метод хорошо заменяет ручное тестирование. Кроме того, можно использовать интеграционные тесты для проверки производительности приложения.

Интеграционный тест, как правило, выполняется на реальном устройстве или эмуляторе, таком как iOS Simulator или Android Emulator.

Приложение изолировано от кода тестового драйвера и запускается после него. Тестовый драйвер позволяет управлять приложением во время теста.

```
group('park-flutter app', () {
FlutterDriver driver;      // драйвер, через который мы подключаемся к устройству
setUpAll(() async {
     driver = await FlutterDriver.connect();            // создаем подключение к драйверу
   });
tearDownAll(() async {
     if (driver != null) {
       driver.close(); // закрываем подключение к драйверу
     }
   });
   test('Имя теста', () async {
     // код теста
   });
   test('Другой тест', () async {
     // код теста
   });
}
```

FlutterDriver взаимодействует с тестируемым приложением через следующие методы:

tap — отправить нажатие виджету;

waitFor — ждать появление виджета на экране;

waitForAbsent — ждать исчезновение виджета;

scroll и **scrollIntoView**, **scrollUntilVisible** — прокрутить экран на заданное смещение или к требуемому виджету;

enterText, **getText** — ввести текст или взять текст виджета;

screenshot — получить скриншот экрана;

requestData — более сложное взаимодействие через вызов функции внутри тестируемого приложения.

Поиск виджетов в интеграционных тестах

в дереве по тексту — **find.text**, **find.widgetWithText**;

по ключу — **find.byValueKey**;

по типу — **find.byType**;

по подсказке — **find.byTooltip**;

по семантической метке — **find.bySemanticsLabel**;

по положению в дереве **find.descendant** и **find.ancestor**.