



Быстрый старт Flutter разработчика

Пошаговое пособие
разработчика кроссплатформенных
мобильных приложений

A photograph of several modern skyscrapers with glass facades, taken from a low angle looking up. The buildings are illuminated at night, with some lights reflecting off the glass. The sky is dark, suggesting it's nighttime.

Андрей Алеев

Flutter and the related logo are trademarks of Google LLC.
We are not endorsed by or affiliated with Google LLC.

Андрей Алеев

Быстрый старт

Flutter-разработчика

http://www.litres.ru/pages/biblio_book/?art=48781701

ISBN 9785005087973

Аннотация

В этой книге даны необходимые элементы, база, которую нужно знать Flutter-разработчику, чтобы писать кроссплатформенные мобильные приложения под Android и iOS на языке Dart. Все это представлено в наглядной форме, на практических примерах, в формате уроков. После их освоения вы сможете именовать себя Flutter-разработчиком.

Flutter and the related logo are trademarks of Google LLC. We are not endorsed by or affiliated with Google LLC.

Содержание

Введение	7
Как работать с этой книгой	9
Урок 1. Запускаем Flutter	13
Кроссплатформенная мобильная разработка	13
Почему Flutter?	15
Настраиваем рабочее окружение	17
Запускаем Hello World! На Android	19
Запускаем Hello World на iOS	26
Урок 2. Язык программирования Dart	30
Введение	30
Переменные, типы и область видимости	32
Видимость	33
Типы	34
final и const	38
Функции	39
Конструкторы	43
Наследование	45
Примеси (mixins)	46
Callable классы	48
Дженерики	49
Асинхронные функции	50
Исключения	52
Использование библиотек	53

Компиляция	55
Урок 3. StatelessWidget и StatefulWidget	58
Все – виджет	58
Состояние виджета	59
StatelessWidget	60
StatefulWidget	63
Типы состояний: Ephemeral и App	68
Урок 4. Создание списка элементов	69
ListView	69
Создание списка через конструктор	71
Импорт пакетов	73
Создание списка с помощью Builder-а	76
Заголовки в списке	78
Урок 5. Загрузка данных с сервера	82
Асинхронность во Flutter	83
Делаем запрос на сервер	85
Http запросы	88
Получение геопозиции	90
Показываем полученные с сервера данные	92
Урок 6. Inherited Widgets, Elements, Keys	98
Inherited widgets	98
Elements	106
Keys	109
Урок 7. Навигация между экранами, Работа с Google Maps	113
Навигация с помощью MaterialPageRoute	113

Интегрируем Google Maps	116
Интегрируем пакет timezone	120
Урок 8. SQLite, Clean Architecture	124
Подключаем SQLite	124
Реализуем паттерн Repository для списка мест	129
Реализуем паттерн Repository для получения погоды	134
Урок 9. BLoC, Streams	136
BLoC	136
Streams	138
Используем BLoC для примера со счетчиком	142
Рефакторим приложение погоды с использованием BLoC библиотеки	146
Урок 10. DI, Тесты	161
Dependency Injection во Flutter	161
Unit тесты	164
UI тесты	171
Интеграционные тесты	175
Заключение	181
Полезные ссылки	183

Быстрый старт Flutter-разработчика

Андрей Алеев

© Андрей Алеев, 2020

ISBN 978-5-0050-8797-3

Создано в интеллектуальной издательской системе Ridero

Введение

Начиная с 2015 года, с момента анонсирования Flutter SDK, популярность этой платформы и языка Dart растет неукоснительно. На популярных профильных ресурсах нарастает количество статей по данной тематике, а многие компании выпускают в магазины приложения, созданные с помощью FlutterTM.

Цель данной книги – научить вас создавать кроссплатформенные мобильные приложения под Android и iOS на Flutter. На практических примерах мы разберем основы языка Dart и базовые принципы построения Flutter-приложений.

Книга будет интересна нативным мобильным разработчикам, которые уже занимаются разработкой приложений, а также всем, кто желает начать писать кроссплатформенные мобильные приложения и познакомиться с языком Dart. Необходимы только базовые знания по программированию. Опыт front-end-разработки придется очень кстати – с ним материал курса будет освоить гораздо проще. Тем не менее, иметь его совсем не обязательно, тем более что после освоения этой книги вы будете на один большой шаг ближе к тому, чтобы именоваться мастером front-end-девелопмента.

Ученые из Оксфордского университета выяснили, что

всего лишь 400 слов покрывают 75% всех английских текстов. Это означает, что со словарным запасом в 400 самых используемых слов вы в трех случаях из четырех будете знать, о чем идет речь в любом тексте. Аналогичным образом написана данная книга: она не претендует на звание учебника или полного справочника платформы Flutter и языка Dart. Мы не будем разбирать по очереди каждый из виджетов в библиотеке material, не будем заучивать все ключевые слова языка Dart. Наоборот, здесь даны самые необходимые элементы, минимум, который надо знать Flutter-разработчику в продакшн, то есть в приложении к решению настоящих задач: *созданию мобильных приложений для реального мира.*

Как работать с этой книгой

Лучше всего усваивается информация, полученная эмпирическим путем. Поэтому ожидается, что вы будете не просто пассивно читать эту книгу, а по каждому уроку напишете код и запустите приложение на двух платформах – Android и iOS.

В идеале, постарайтесь написать свое приложение, которое будет, к примеру, загружать фотки котиков из сети или выполнять более утилитарную задачу, пусть калькулятор. На ваш вкус. В этой книге мы будем разбирать два примера – сначала создадим простой счетчик, а затем более сложный – загрузка прогноза погоды с сайта openweathermap.org. Если вы захотите написать такое же приложение, вам потребуется API KEY с их сайта, а также API KEY Google Maps. Помимо этого, желательно иметь опыт работы с Git, Android Studio, Gradle.

Всего в книге 10 глав-уроков, первые уроки более простые, последние – более сложные, и для них, возможно, потребуется больше времени. Помогать вам будет уже написанный и работающий код в репозитории проекта – https://github.com/acinonyxjubatus/flyflutter_fast_start – FlyFlutter Fast Start на гитхабе, там для каждого урока выделена своя ветка. Страйтесь не просто копировать оттуда код, а вдумчиво писать его, только лишь сверяясь с ко-

дом на гитхабе. Ниже вкратце приведено описание уроков, а также указаны ссылки на соответствующие ветки репозитория.

Урок 1. Запускаем Flutter [ветка *lesson_1_hello_world*]

Научимся запускать проект на Flutter под Android и iOS, а также совершать простейшие манипуляции с виджетами. Помимо этого, узнаем чем может быть полезен Flutter и когда на нем можно создавать приложения.

Урок 2. Язык программирования Dart

Обзорно пройдемся по основным возможностям и правилам языка Dart

Урок 3. StatelessWidget и StatefulWidget [ветки *lesson_3_1_stateless_widget*, *lesson_3_1_stateful_widget*]

Научимся создавать Stateless и StatefulWidget-ы. Узнаем про состояния виджетов, попробуем ими манипулировать. Также узнаем, как декорировать и выравнивать виджеты.

Урок 4. Создание списка элементов [ветка *lesson_4_listview*]

Познакомимся с ListView, узнаем какие есть способы его создания. Полученные знания применим для создания списка с прогнозами погоды.

Урок 5. Загрузка данных с сервера [ветка *lesson_5_http*]

Узнаем как можно выполнить асинхронную работу во Flutter. Сделаем запрос на сервер, получим, распарсим и покажем полученную информацию на клиенте. Таким об-

разом, создадим полноценное клиент-серверное приложение.

Урок 6. Inherited Widgets, Elements, Keys [ветка *lesson_6_inherited*]

Узнаем, что такое Inherited Widget, а также на примере посмотрим как он работает. Разберемся с тем, что такое Element-ы и как они работают. Помимо этого, мы познакомимся с ключами Keys и узнаем когда и как их нужно использовать.

Урок 7. Навигация между экранами, Работа с Google Maps [ветка *lesson_7_navigation_maps*]

Научимся переключать экраны с помощью Navigator-a. Сможем подключить и показать карты от Google Maps в приложении, а также подключим дополнительный необходимый в примере пакет timezone.

Урок 8. SQLite, Clean Architecture [ветка *lesson_8_sqlite_clean_architecture*]

Сумеем подключить SQLite и сохранить данные в локальной базе данных, а также прочесть их. Убедимся, что во Flutter тоже можно и нужно писать чистый код и напишем свою реализацию паттерна Repository.

Урок 9. BLoC, Streams [ветки *lesson_9_bloc*, *lesson_9_1_counter_bloc*]

Узнаем, что такое BLoC, чем он полезен и как использовать библиотеку bloc. Все это применим на практике: мы

произведем значительный рефакторинг приложения погоды, придав коду приличествующий вид – повысим читаемость и поддерживаемость.

Урок 10. DI, Тесты [ветки *lesson_10_di_tests*, *lesson_9_1_counter_bloc*]

Освоим технику инверсии зависимостей применительно к Flutter разработке. На практическом примере реализуем паттерн Dependency Injection во Flutter в примере приложения погоды. Затем узнаем, какие бывают тесты. Напишем unit-тесты, widget (UI-тесты) и интеграционные тесты для приложения с погодой.

Урок 1. Запускаем Flutter

В этой главе:

- Кроссплатформенная мобильная разработка
- Почему Flutter?
- Настраиваем рабочее окружение
- Запускаем Hello World на Android
- Запускаем Hello World на iOS

Кроссплатформенная мобильная разработка

Для начала несколько слов о том, что такое Flutter и зачем он нам нужен. Если вы знаете ответ на вопрос, что такое кроссплатформенная разработка и Flutter, листайте дальше к пункту 3 этой главы: «Настройка рабочего окружения».

Так вышло, что на сегодняшний день в мире мобильных устройств лидируют 2 платформы – iOS от Apple и Google Android. Представьте, что вам прямо сейчас надо написать мобильное приложение под обе операционные системы. Вам нужно нанять, условно, по 1—3 программиста на каждую платформу. Или по 5, или по 7, в зависимости от сложности проекта.

Возьмем число 5 на платформу – оптимальное, на мой

взгляд, количество для проекта средней сложности. Это означает 10 программистов в сумме. Из них статистически будет 2—4 очень хороших, сильных программиста, 2—4 слабеньких и 2—4 средних по уровню. Если же язык программирования один и кодовая база одна, значит, можно взять из этих же 10 программистов 5 лучших. Конечно, останется 5 программистов не у дел, но это возможность переместить их на другие участки работы или дополнительный стимул привести им профессионально. Иными словами, слив скопированных работ до одной кодовой базы, можно одновременно уменьшить расходы на разработку и увеличить качество. Конечно, это все теория. На практике большинство выбирают нативную разработку, и зачастую оправдано, поскольку только она дает максимальное качество конечного продукта. Но зачастую – не значит всегда. Рассмотрим, когда и как можно применить Flutter.

Почему Flutter?

Если Вы думаете, стоит ли Вам браться за кроссплатформу и конкретно за Flutter, ответьте себе на вопрос: зачем нам нужно это приложение, какие бизнес-цели мы с помощью него решаем? Сравните свой ответ с двумя абзацами ниже и решите к какому относится ваше приложение в большей степени.

Для начала определим, в каких случаях Flutter не очень хорошо подходит. Если кратко, то это все кейсы, когда приложение *представляет собой конечный продукт* и будет конкурировать с другими такими продуктами в магазине приложений за топовые позиции. Например, это может быть новая Angry Birds, рисовалка, читалка, фитнес-приложение. Вам нужна будет максимальная скорость, точность и плавность при работе приложения, и это все на сегодняшний день дает только нативное приложение. Также следует выделить категорию приложений, в которых планируется активно использовать встроенные в устройства датчики, такие как Bluetooth, гироскопы, камеру. Это конечно, не значит, что Flutter нельзя использовать в перечисленных случаях. Но высока вероятность, что вам так или иначе придется писать нативный код и/или костыли.

С другой стороны, существует множество кейсов, когда реальный бизнес желает получить мобильное приложение,

которое будет помогать им в реализации бизнес-процессов и/или дополнять их, но без фанатичной погони за самым модным UI и супер-быстродействием. В качестве примера можно привести программы лояльности, мобильное рабочее место для сотрудников, интернет-магазин, а также многие другие, где приложение будет *обслуживать реальный бизнес-процесс*.

Резюмируя, небольшие приложения с оффлайновым бизнесом можно и нужно создавать на Flutter, а сам framework рекомендуется к изучению всем мобильным разработчикам.

Настраиваем рабочее окружение

Теперь, когда мы разобрались, в каких случаях мы можем использовать Flutter, давайте уже научимся им пользоваться!

Для начала установим Flutter SDK. Скачайте архив с SDK с [официального сайта](https://flutter.dev/docs/get-started/install) (<https://flutter.dev/docs/get-started/install>). Выберите вашу платформу (Windows, Mac, Linux) и следуйте инструкции.

После распаковки архива добавьте в PATH Flutter/bin
export PATH=\$PATH:`pwd`/flutter/bin // Mac

Здесь может потребоваться перезапустить компьютер.

После установки в командной строке запустите команду
flutter doctor

и убедитесь, что у вас все установлено корректно.

Если планируете собирать и тестировать под iOS, то необходимо установить и обновить Xcode и соответствующие пакеты с помощью brew, следя подсказке в ответе flutter doctor, а также следовать [инструкции для macos](https://flutter.dev/docs/get-started/install/macos) <https://flutter.dev/docs/get-started/install/macos>)

```
✓) Android toolchain - develop for Android devices (Android SDK Version 29.0.1)
(!) iOS toolchain - develop for iOS devices (Xcode 10.3)
  ✘ libimobiledevice and iDeviceInstaller are not installed. To install with Brew, run:
    brew update
    brew install --HEAD usbmuxd
    brew link usbmuxd
    brew install --HEAD libimobiledevice
    brew install iDeviceInstaller
  ✘ ios-deploy not installed. To install:
    brew install ios-deploy
  ✘ CocoaPods not installed.
    CocoaPods is used to retrieve the iOS platform side's plugin code that responds to
    your plugin usage on the Dart side.
    Without resolving iOS dependencies with CocoaPods, plugins will not work on iOS.
    For more info, see https://flutter.dev/platform-plugins
  To install:
    brew install cocoapods
    pod setup
[!] Android Studio (version 3.5)

```

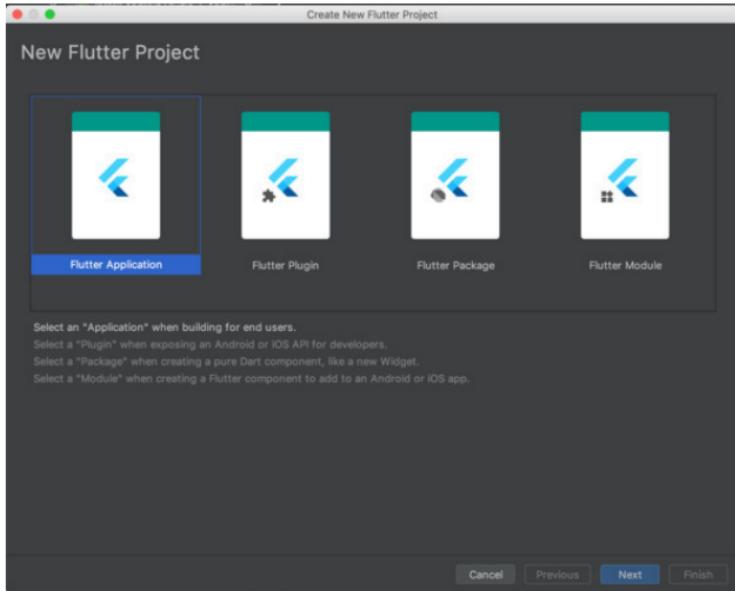
Результаты команды flutter doctor с ошибками

Если планируете тестировать на Android-устройстве, то используйте Android Studio. Если у вас нет Android Studio, следуйте инструкции по установке (<https://developer.android.com/studio/install>), чтобы установить ее.

Запускаем Hello World! На Android

Итак, приступим к созданию первого приложения на Flutter. Для этого курса вы также можете использовать Android Studio, XCode или VS Code – как вам удобно. Мы будем рассматривать на примере Android Studio.

Запустите Android Studio и выберите *Start a new Flutter project*.



Интерфейс создания нового проекта

Выберите **Flutter Application**

Заполните имя *flutter_hello_world* в поле **Project Name**
company domain – **flyflutter.ru** – и жмем Finish.

После запуска мы сразу видим открытый файл **main. dart**
В нем – видим строчку

```
void main () => runApp (MyApp ());
```

это начальная точка приложения. Функция **main ()** – это стартовая точка всех приложений на языке Dart. В ней мы здесь вызываем конструктор класса **MyApp**, который наследуется от **StatelessWidget** – это тип UI компонента – виджета. Подробнее про язык Dart мы поговорим во второй лекции, а про виджеты – в третьей.

Итак, слева мы видим дерево проекта, справа – редактор.

The screenshot shows the Android Studio interface. On the left, the Project tool window displays the project structure: flutter_hello_world (root), lib (containing main.dart), test, ignore, metadata, build, android (containing flutter_hello_world_android), build.gradle, build.gradle.kts, flutter_hello_world.iml, pubspec.lock, pubspec.yaml, README.md, External Libraries, and Scratches and Consoles. The main.dart file is selected in the Project tree and is open in the code editor on the right. The code editor shows the following code:

```
import 'package:flutter/material.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: Text('This is Flutter'),
        ),
        body: Center(
          child: Text('Hello World!'),
        ),
      ),
    );
}
```

Код **main. dart** только что созданного проекта

Весь общий для Android и iOS код находится в папке lib.

Сейчас у нас там только файл main.dart

Android Studio сгенерировала простую логику инкрементирования счетчика, мы ее пока удалим, чтобы она нас не путала, и заменим на более простой вариант

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: Text('This is Flutter'),
        ),
        body: Center(
          child: Text('Hello World!'),
        ),
      ),
    );
  }
}
```

Жмите на иконку молнии – **Hot Reload** – для применения изменений.

The screenshot shows the Android Studio interface with the project 'flyflutter_fast_start' open. The top bar displays the device as 'Pixel 2 API 28'. The main window shows the 'AndroidManifest.xml' file, which contains the following XML code:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.flyflutter.flyflutter_fast_start">
    <!-- io.flutter.app.FlutterApplication is an application
         that calls FlutterMain.startInitialization(this); --
    In most cases you can leave this as-is, but
    if you need some additional functionality it is fine to subclass
    FlutterApplication and put your custom class here.
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Надо отметить, что Hot Reload во Flutter работает действительно быстро и значительно сокращает время разработки.

Ура, на экране вы должны увидеть «Привет, Мир!».



«Привет, Мир!» на эмуляторе

Рассмотрим код подробнее. Как уже говорилось выше, MyApp наследуется от StatelessWidget, это неизменяемый UI компонент-виджет. Вообще, все во Flutter – это виджеты, и приложение тоже. В виджете мы переопределяем метод build, в котором указывается, что и как отрисовать.

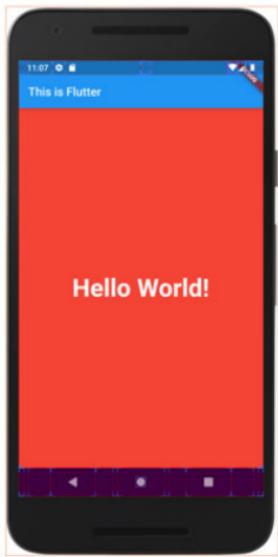
В нашем примере мы возвращаем объект MaterialApp, который создаем посредством конструктора. А в конструктор передаем название, тему и виджет home, которому назначаем Scaffold – скелет приложения, который в свою очередь содержит appBar и body. Здесь уместна аналогия с HTML,

где также есть тэги <title> и <body>.

Давайте немного увеличим текст и поиграем цветами:

```
home: Scaffold(  
    backgroundColor: Colors.red,  
    appBar: AppBar(  
        title: Text('This is Flutter'),  
    ),  
    body: Center(  
        child: Text('Hello World!',  
            style: TextStyle(  
                fontSize: 42.0, // делаем текст большим  
                fontWeight: FontWeight.bold, // жирным  
                color: Colors.white, // белым  
            )  
        ),  
    ),  
,
```

Виджету Scaffold мы задали красный фон, а виджету текста применили стиль, чтобы сделать его больше и заметнее.

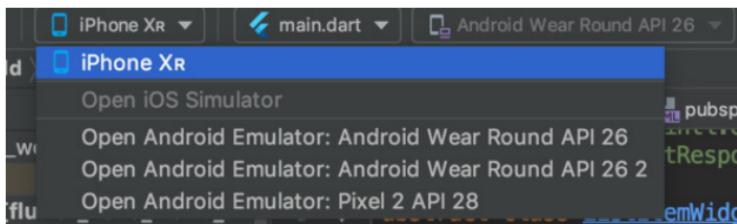


Привет, Мир! на Андроид

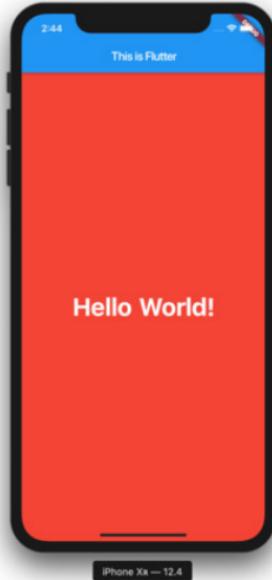
Преимущество Flutter в том, что вся логика работы с внешним видом приложения (UI) прописывается в коде на том же языке, что и бизнес-логика – на dart. Нет необходимости залезать в папку с ресурсами и редактировать xml верстку.

Запускаем Hello World на iOS

Мы же пишем кроссплатформенный код! Давайте запустим созданное приложение на iOS-девайсе. Для этого просто выберите подключенный iOS-девайс или эмулятор в dropdown-списке и нажмите «Запустить».



Выбор эмулятора iOS



Привет, Мир! на iOS

Экран выглядит потрясающе, однако вверху экрана мешается ненужная иконка debug, да и иконка приложения сейчас никакая. Исправим это.

Для того, чтобы убрать ленточку debug, в добавьте в MaterialApp флаг *debugShowCheckedModeBanner* со значением false

```
return MaterialApp (  
  debugShowCheckedModeBanner: false,
```

Чтобы поменять иконку, нужно добавить в pubspec.yaml пакет

```
dev_dependencies:  
  flutter_launcher_icons: ^0.7.4
```

Этот пакет значительно упростит нам добавление иконки для двух платформ сразу. Добавим теперь в корне проекта папку assets с иконкой, а также пропишем путь к иконке

```
flutter_icons:  
  android: «launcher_icon»  
  ios: true  
  image_path: «assets/icons/flyflutter_ic_512.webp»
```

не забудьте сказать flutter, чтобы смотрел папочку assets

```
flutter:
```

```
uses-material-design: true  
assets:  
  - assets/  
  - assets/icons/
```

После этого для генерации иконок запустите в терминале команды

```
flutter pub get
```

```
flutter pub run flutter_launcher_icons: main
```

Чтобы поменять лейбл (название иконки) приложения:

Для Android – найдите манифест в android/app/src/main/AndroidManifest.xml и добавьте в тег application строку

android: label=«FlyFlutter»

Для iOS же зайдите в Info.plist по пути ios>runner/Info.plist и для ключа укажите имя **CFBundleName**

<key> CFBundleName </key>
<string> FlyFlutter </string>

Готово. Запустите снова для проверки.

Урок 2. Язык программирования Dart

В этой главе:

- Переменные, типы и область видимости
- Функции
- Конструкторы
- Наследование
- Примеси (*mixin*)
- Callable классы
- Дженерики
- Асинхронные функции
- Исключения
- Использование библиотек
- Компиляция

Введение

Приложения под Flutter пишутся на языке Dart. Даже сам фреймворк написан на нем. Dart – это высокоуровневый объектно-ориентированный язык программирования общего назначения с открытым исходным кодом. Был разработан в Google. Испытал влияние C, Javascript, C#, Java. В нем также как и в Java и C# присутствует garbage collector. Язык поддерживает интерфейсы, примеси (англ. Mixin), абстракт-

ные классы, дженерики и статическую типизацию.

Dart был представлен публике в 2011 году авторами Ларсом Барком (Lars Bark) и Каспером Лундом (Kasper Lund). Релиз версии 1.0 состоялся в 2013 году, а версии 2.0 в 2018

Примечание: Здесь и далее в этом курсе мы рассматриваем Dart версии 2

Все приложения на Dart, как и на C и в Java, имеют точку входа в функции main ()

```
void main() {  
    print('Hello, World!');  
}
```

В случае, если необходимо запустить программу на Dart из командной строки, то можно использовать параметризованную main:

```
void main(List<String> args) {  
    print(args);  
}
```

Переменные, типы и область видимости

Dart типобезопасный язык. В нем используется как статическая типизация на этапе компиляции, так и динамическая проверка во времени исполнения (runtime) программы. Несмотря на наличие статической типизации, указывать тип переменной необязательно. Например, все объявления и инициализации ниже корректные:

```
var name = 'Dart';
var year = 2011;
String author;
author = "Lars Bark";
List<Foo> myList = <Foo>[];
List<Foo> oldList = new List();
```

Примечание: В Dart 2 ключевое слово `new` стало необязательным

Видимость

По умолчанию, все переменные имеют публичную область видимости. Таких привычных для Java-программистов ключевых слов, как `private`, `protected` и `public` в Dart нет.

Однако если добавить нижнее подчеркивание `_` к имени переменной, такая переменная будет иметь область видимости библиотеки, в которой она находится.

Типы

Все объекты в Dart наследуются от базового типа Object. Это аналог Object в Java. В нем также есть метод hashCode () и аналог equals, который заменяет оператор сравнения ==

Так же в классе Object присутствует метод toString ()

Встроенные типы включают:

- Числовые (num и его наследники int и double)
- Строковые (strings)
- Булевы (Booleans)
- Списки, или массивы (list)
- Сеты (set)
- Мапы (map)
- Руны (for expressing Unicode characters in a string)
- Символы (symbols)

int – Целочисленные переменные. На виртуальной машине Dart диапазон составляет от -2^{63} до $2^{63}-1$

Примечание: При компиляции в JavaScript диапазон int-а -2^{53} до $2^{53}-1$

double – 64-битные числа с плавающей запятой

И **int** и **double** наследуются от типа **num**

String

Строковые переменные в Dart представляют собой последовательности из UTF-16 символов. Для инициализации можно использовать как двойные, так и одинарные кавычки:

```
var s1 = 'Строка в одинарных кавычках';
String s2 = "Строка в двойных кавычках";
```

Значения переменных можно использовать в строках с помощью конструкции `$ {выражение}`

```
var a = 2;
var b = 2;
var s = '$a + $b = ${a+b}';
// получим «2+2=4»
```

bool

Для создания булевых переменных в Dart существует ключевое слово `bool`. При инициализации можно использовать литералы `true` и `false`. То есть, инициализация `bool b = 0;` – некорректна, правильно

```
bool b = true;
```

List

Списки – это коллекции проиндексированных объектов.

Примеры объявления и инициализации списков:

```
List<int> list1 = new List();  
List<int> list2 = List();  
var list3 = [1, 2, 3];
```

Для инициализации в Dart 2.3 добавлен спред оператор – троеточие – с помощью него можно добавить в список множество значений:

```
var list = [1, 2, 3];  
var list2 = [0, ...list];
```

Sets

Сеты – это неупорядоченные наборы уникальных элементов. В Dart для того, чтобы создать сет, нужно использовать фигурные скобки для непустого набора и фигурные скобки в сочетании с угловыми и типом объектов для пустого:

```
var colors = {'red', 'green', 'blue'}; // сразу инициализируем сет  
var colors = <String>{}; // объявляем пустой сет  
Set<String> colors = {} // тоже корректно
```

Maps

Мапы – это наборы данных в формате ключ-значение. Ключами, как и значениями, могут быть объекты любых типов. Каждый ключ является уникальным, значения могут быть разными, а могут дублироваться. Посмотрим на примере:

```
var ballGames = {  
  'baseball': 'club',  
  'basketball': 'hands',  
  'football': 'foots'  
};
```

Альтернативные способы инициализации

```
var ballGames = Map();  
ballGames['baseball'] = 'club';  
***  
var ballGames = Map();  
ballGames[2] = 'hands';
```

Runes

Dart поддерживает руны – спецсимволы юникод. Используйте, если хотите добавить смайлики. Попробуйте запустить в [dartpad](#)

```
Runes input = new Runes('\u{1f60e}');  
print(new String.fromCharCodes(input));
```

final и const

В языке также присутствуют ключевые слова `final` и `const`.

Если переменную не планируется изменять, то следует задать ей модификатор `final` перед типом или словом `var`. Такая переменная может быть проинициализирована единожды. Переменные `const` неявно считаются `final`. Такие переменные используются для задания констант на этапе компиляции.

Функции

В Dart даже функции являются объектами. Это значит, что функции можно назначать переменным и передавать в качестве аргументов в другие функции. Тип возвращаемого значения указывается перед именем функции. Делать это необязательно, хотя и рекомендуется:

```
int doubleIt(int value) {  
    return value * 2;  
}  
  
doubleIt(value) { // корректно  
    return value * 2;  
}
```

Поскольку эта функция содержит всего одно выражение, ее можно укоротить до одной строчки:

```
int doubleIt(value) => value * 2;
```

Оператор `=>` -это сокращение фигурных скобок и слова `return`.

Опциональные параметры

При объявлении функции мы можем в ее сигнатуре указать значения по умолчанию. Например, нам понадобится вызывать какую-то функцию много раз с одним и тем же параметром, но при этом необходимо сохранить гибкость. В таком случае, при вызове функции с параметром по умолчанию его (этот параметр) можно не указывать.

В Dart существует два типа опциональных параметров: позиционные и именованные. Рассмотрим их подробнее.

Примечание: Опциональный параметр не может быть одновременно и позиционным и именованным

Именованные – такие параметры при вызове функции можно указывать опционально с именем и последующим двоеточием и оборачиванием в фигурные скобки. Пример:

```
updateWidget(int position, { bool withTitle: true, int padding:  
8})
```

Аннотация @required делает этот параметр обязательным

Позиционные – опциональные параметры, помещаемые при объявлении функции в квадратные скобки. Они будут читаться при вызове по их позиции среди аргументов.

```
String something(String parametr1, [String parametr2]) {  
    if (parametr2 != null) {  
        return '$parametr1 and $parametr2';  
    } else {  
        return '$parametr1';  
    }  
}
```

Функции как объекты

Функции можно передавать в качестве параметров другим функциям, а также назначать их переменным:

```
void doubleIt(int it) {  
    it = it * 2  
    print(it);  
}  
var list = [1, 2, 3];  
// передаем на выполнение в foreach функцию  
list.forEach(doubleIt);  
  
// функция как переменная  
var doubleIt = (par1) => '${par1 * 2}';  
// функция как переменная с указанием типа  
Function tripleIt = (par1) => '${par1 * 3}';
```

Анонимные функции (лямбды)

Выше мы уже увидели пример *анонимной функции* – это функция не имеющая имени.

```
var list = ['a', 'b', 'c'];  
list.forEach ((it) {  
    print('$it is ${list.indexOf(it) + 1} letter of alphabet');  
});
```

В этом примере у нас всего одно выражение в фигурных

скобках, поэтому такую запись можно сократить до

```
list.forEach ((it) => print('${list.indexOf(it) + 1} letter  
of alphabet));
```

Примечание: Переопределение методов (функций)
родительского класса выполняется с помощью
аннотации @override

Конструкторы

Конструкторы позволяют создать объект такого же типа, как и класс, в котором они объявлены. Выглядят они как функции:

```
class Rectangle {  
    num width, height;  
  
    Rectangle(num width, num height) {  
        this.width = width;  
        this.height = height;  
    }  
}
```

Аналогично Java, если не объявлять конструктор, то будет использоваться конструктор по умолчанию – без параметров. Однако, в отличие от Java, конструкторы не наследуются.

Конструкторы могут быть именованными:

```
class Rectangle {  
    num width, height;  
  
    Rectangle(this.x, this.y)  
  
    Rectangle.square() {  
        this.width = 10;  
        this.height = 10;  
    }  
}
```

При наследовании классов конструкторы родительские конструкторы можно вызывать с помощью конструкции: **super**

```
class Rectangle {  
    num width, height;  
    Rectangle.fromParams(Map params) {  
        } ***  
    }  
  
class Square extends Rectangle {  
    Square.fromParams(Map params) : super.fromParams() {  
        } ***  
    }
```

Наследование

Наследование выполняется с помощью ключевого слова `extends`

В Dart отсутствует ключевое слово `interface`. Вместо этого всякий класс неявно представляет собой интерфейс (абстрактное поведение или набор характеристик), который затем можно имплементировать в других классах.

```
// Музыкант. Скрытый интерфейс содержит метод play()
class Musician {

    // В интерфейсе
    final instrument;
    // Не в интерфейсе – это конструктор
    Musician(this.name);
    // В интерфейсе
    void play() => 'Hi, I can play $instrument';
}

// Гитарист имплементирует Музыканта
class Guitarist implements Musician {
    get name => 'Guitar';
    void play() => 'Hi, I can play $instrument';
}
```

Примеси (mixins)

Примесь, или **Mixin** в языке Dart – это класс, описывающий некоторое поведение. Он чем-то напоминает интерфейс, однако правила его использования несколько отличаются. Примеси не наследуются, а как бы подключаются, «примешиваются» к коду класса, поэтому и называются примесями. Отличие от интерфейса в Java заключается в том, что методы примеси уже не надо переопределять. Посмотрим на примере.

Предположим, нам нужно описать музыканта, который умеет играть разные стили музыки.

```
// Гитарист имплементирует Музыканта и умеет играть разные стили
class Guitarist implements Musician with Jazz, Rock, Funk {
    get name => 'Guitar';
    void play(String arg) => 'Hi, I can play $arg by $instrument';
}

// примесь
mixin Rock {
    bool knowsHowToPlayACDC = true;

    void playPopMusic() {
        if (knowsHowToPlayACDC) {
            play("TNT")
        }
    }
}
mixin Jazz {
    bool knowsHowToPlayEllington = true;

    void playJazzMusic() {
        if (knowsHowToPlayEllington) {
            play("Take the A Train")
        }
    }
}
// примесь Funk
mixin Funk {
    bool knowsHowToPlayBrown = true;

    void playFunkMusic() {
        if (knowsHowToPlayBrown) {
            play("I FEEL GOOD!")
        }
    }
}
```

Callable классы

Объекты класса можно вызывать как функции, если имплементировать в них метод **call ()**

```
class ClassAsFunction {
    call(int a, int b) => a*b;
}
void main() {
    var classAsFunction = ClassAsFunction();
    var out = classAsFunction(2, 2); // вызываем класс как функцию
    print('$out'); // получим 4
}
```

Дженерики

В Dart присутствует поддержка Generics. Работают они аналогично Java, и применяются в повторно-используемых компонентах, например, в абстракциях:

```
abstract class Foo<T extends BaseClass> {  
    ***  
}
```

Если вы посмотрите исходники класса List, то увидите, что он уже содержит дженерики, поэтому в угловых скобках мы указываем тип переменных

```
var names = List<String>();  
names.addAll(['Mercury', 'Venus', 'Earth']);  
names.add('Mars'); // можно  
names.add(42); // ошибка типа
```

Асинхронные функции

Для выполнения фоновой работы в Dart есть Future и Stream объекты

Future

Future <T> – это объект, представляющий собой асинхронную операцию, которая вернет после выполнения объект типа Т. Когда вызывается функция, возвращающая Future, происходят последовательно два действия:

- Эта функция ставит в очередь работу, которую она должна выполнить, и сразу же возвращает невыполненный объект Future
- Когда операция выполнена, объект Future завершается с ошибкой или полученным значением

Для того, чтобы написать асинхронную функцию, ее нужно пометить как `async`, а саму асинхронную работу словом `await`:

```
import 'dart:async'; // стандартная библиотека для асинхронной работы
Future<void> updateData() async {
    var digest = await getDataFromServer(); // дальше этой строчки мы не
    // уйдем, пока не получим результат или ошибку
    print(digest);
}
```

Примечание: функция `getDataFromServer ()` тоже

должна возвращать Future

Stream

Если вы знакомы с RxJava, то вам все должно быть понятно из названия. Потоки – это последовательности (Iterable) асинхронных событий. Рассмотрим на простом примере:

```
Future<int> sumStream(Stream<int> stream) async { // получаем на
    вход потом целых чисел
    var sum = 0;
    await for (var value in stream) {
        sum += value;
    }
    return sum;
}

Stream<int> countStream(int to) async* {
    for (int i = 1; i <= to; i++) {
        yield i; // испускаем элемент
    }
}

main() async {
    var stream = countStream(10);
    var sum = await sumStream(stream);
    print(sum); // в итоге получим 55
}
```

Слово **yield** испускает элемент в функции *countStream*, которая возвращает Stream, то есть представляет собой поток данных. Затем мы каждое новое значение из этого потока прибавляем к сумме предыдущих в функции *sumStream*.

Исключения

Если вы знакомы с исключениями по Java, то вкратце – все исключения в Dart unchecked. Иначе говоря, все исключения в Dart происходят в Runtime-е, то есть могут быть выброшены во время исполнения программы.

Функция, которая может выбросить исключение, не обязана объявлять об этом в своей сигнатуре, а использовать эту функцию, соответственно, не обязательно в блоке try catch.

Еще одно отличие от Java в том, тип исключения указывается после слова **on**

```
void getException(){
    throw Exception('bam!');
}

try {
    getException();
} on Exception {
    // do something
} catch (e) {
    print('Произошла ошибка: $e');
} finally {
    // После слова finally код выполнится обязательно
    closeDatabase();
}
```

При отлавливании исключений можно использовать и **on**, и **catch**, и оба одновременно. **catch** позволяет получить доступ к объекту exception.

Использование библиотек

Чтобы импортировать библиотеку, пространство имен или класс, необходимо в верхней части файла прописать путь к ним после слова **import**

```
import 'package:shapes/geometric.dart';
import 'package:abstracts/colored.dart' as colored;

/// Используем Rectangle из библиотеки geometric.
Rectangle rectangle1 = Rectangle();

// используем Rectangle из библиотеки colored.
colored.Rectangle rectangle2 = colored.Rectangle();
```

При использовании указанного выше способа загрузки библиотек, они загружаются сразу. Чтобы они загружались по требованию, им можно добавить модификатор **deffered as**

```
import 'package:abstracts/colored.dart' deffered as colored;
```

И затем в нужный момент загрузить с помощью функции *loadLibrary()*:

```
Future greet() async {
  await colored.loadLibrary();
  hello.printGreeting();
}
```

Компиляция

Написанный на Dart код нужно скомпилировать под целевую платформу. Поскольку Dart – это язык общего назначения, он может компилироваться как по паттерну JIT (Just In Time), так и по паттерну AOT (Ahead Of Time).

Отличие JIT от AOT в том, что при JIT код компилируется непосредственно перед использованием. Пример JIT – это JavaScript и движок V8 Chromim-a. При этом мы не зависим от архитектуры платформы, однако код может компилироваться долго в рантайме, что ощутимо замедлит быстродействие.

В случае с AOT мы заранее компилируем весь код проекта и получаем бинарный файл. Пример – это C++, Java (JVM). На выходе мы получаем быстродействующее приложение, но под одну целевую платформу, под другую платформу нужен уже другой бинарник.

Как работает Flutter? В зависимости от способа сборки – по-разному. В случае сборки release iOS используется AOT. Для release Android может использоваться как AOT, так и CoreJIT (вариация JIT).

При разработке мы хотим видеть изменения быстро, и у нас есть такая возможность – Hot Reload. Он как раз-таки и работает по паттерну JIT. Платформа формирует некие снимки состояния – snapshots, которые затем переиспользу-

ЮТСЯ.

Pattern / Term	Compilation Pattern	Architecture Specific	Package Size	Dispatch Dynamically
Script	JIT	False	Small	True
Script Snapshot	JIT	False	Smallest	True
Application Snapshot	JIT	True	Lager	True
AOT	AOT	True	Lagest	False

Типы snapshot-ов

Term / Platform	Android	iOS
Stage	debug	debug
Compilation Pattern	Kernel Snapshot	Kernel Snapshot
Packaging Tool	dart vrm (2.0)	dart vrm (2.0)
Command	flutter build bundle	flutter build bundle
Packed Product	flutter_assets/*	flutter_assets/*

Использование snapshot-ов в debug режиме

Term / Platform	Android	iOS	Android(-build-shared-library)
Stage	release	release	release
Compilation Pattern	Core JIT	AOT Assembly	AOT Assembly
Package Tool	gen_snapshot	gen_snapshot	gen_snapshot
Flutter Command	flutter build aot	flutter build aot -ios	flutter build aot -build-shared-library
Packed Product	flutter_assets/*	App.framework	app.so

Использование snapshots в release режиме

Итоги обзора

Dart – мощный инструмент разработки, впитавший в себя многое от Java, Javascript и других языков, успешно применяемых в промышленной разработке. Перейти на Dart программистам, писавшим ранее на ООП-языках не составит труда, а новичкам изучить его несложно. На текущий момент Dart уже можно применять в релизных продуктах:

- Flutter – для мобильных приложений
- AngularDart и Hummingbird для web-разработки
- Aqueduct для бэкенда

Урок 3. StatelessWidget и StatefulWidget

В этой главе:

- Все – виджет
- Состояние виджета
- StatelessWidget
- StatefulWidget
- Типы состояний: *Ephemeral* и *App*

Все – виджет

Начнем с ответа на вопрос «Что такое Widget во Flutter?». Виджет – это основной строительный блок пользовательского интерфейса приложений. Причем, эти блоки, как матрешки, можно вкладывать один в другой, образуяложенную иерархичную структуру. Таким образом все во Flutter – это виджет; начиная от текста на кнопке, заканчивая самим приложением, которое тоже является виджетом. Главное, что нужно сразу понять про виджеты, это то, что по типу они разделяются на две основные категории: **Stateless** и **Stateful**. Если вы знаете английский, то из названия Вы можете сразу понять в чем их отличие. Одни имеют состояние, другие – нет. Давайте копнем поглубже.

Состояние виджета

Состояние – это «любая информация, необходимая для отрисовки UI в любой момент времени». Будучи Java и Kotlin разработчиком, я привык писать в императивном стиле. Например, `textView.setText («Lorem»)` или `textView.text=«Lorem»`. То есть мы меняли внешний вид виджета напрямую, прямо указывая системе что надо поменять.

Поскольку Flutter декларативный, пользовательский интерфейс строится как некоторая функция от состояния:

$$UI = f(state)$$

Другими словами, UI наблюдает за состоянием, и если вы хотите изменить UI, Вам нужно обновить состояние. И здесь на сцену выходят два типа виджетов – те, которым можно менять состояние в рантайме, и те, которым нельзя. Здесь уместно провести аналогию с `val` и `var` переменными в Kotlin. Рассматривайте `StatelessWidget` как `val`, а `StatefulWidget` как `var` переменную.

StatelessWidget

Stateless виджет не может менять свое состояние, то есть он иммутабелен. Такой тип виджетов удобно использовать для статичных элементов экрана, которые надо отрисовать один раз и не трогать больше. Например, это могут быть заголовки, label-ы, иконки, изображения из локальных ресурсов и т. п. Самый яркий пример – это само приложение. Оно наследуется от класса **StatelessWidget**. Однако, давайте создадим для тренировки свой StatelessWidget. Пусть это будет простой label с каким-то задаваемым текстом.

```
class HintLabel extends StatelessWidget {
    final String text;

    const HintLabel(this.text);

    @override
    Widget build(BuildContext context) {
        return DecoratedBox(
            decoration: BoxDecoration(color: Colors.amber[200]),
            child: Padding(
                padding: const EdgeInsets.all(8.0),
                child: Text(text,
                    style: TextStyle(color: Colors.grey[700])),
            ),
        );
    }
}
```

Вы можете заметить, что поле `text` – `final`, а конструктор с модификатором `const`. Таким образом мы делаем текст и сам инстанс объекта неизменяемым. В методе `build` рисуем виджет `DecoratedBox` – это специальный виджет, кото-

рый наследуется от SingleChildRenderObjectWidget, то есть от виджета с одним child-ом. Он используется для задания внешнего вида виджету. Мы это делаем указывая в child-e BoxDecoration с цветом фона. Обратите внимание на использование цветовой гаммы Material. Мы берем цвет amber 200. Подробнее про цвета вы можете узнать на странице <https://material.io/guidelines/style/color.html>

Давайте теперь используем наш новый UI компонент и добавим его на главный экран

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'StatelessWidget sample',
      theme: ThemeData(
        primarySwatch: Colors.amber,
      ),
      home: Scaffold(
        backgroundColor: Colors.amber[300],
        appBar: AppBar(
          title: Text('Labels'),
        ),
        body: Center(
          child: Column(
            mainAxisSize: MainAxisSize.min,
            mainAxisAlignment: MainAxisAlignment.center,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
              HintLabel('custom label 1'),
              SizedBox(height: 8.0),
              Text('text widget'),
              SizedBox(height: 8.0),
              HintLabel('custom label 2')
            ],
          ),
        ),
      );
  }
}
```

Запускаем с помощью Hot Reload и смотрим на наши

label-ы



Кастомные виджеты-лейблы

Получилось симпатичненько. Чтобы сверится, откройте ветку *lesson_3_1_stateless_widget* Давайте теперь создадим виджет с изменяемым состоянием и посмотрим как он работает.

StatefulWidget

Stateful виджеты можно менять во времени исполнения, то есть они мутабельны. Их следует использовать в случае с вводом текста, слайдером, чекбоксами и т. п. Чтобы создать новый мутабельный виджет нужно отнаследоваться от класса StatefulWidget и создать класс State (состояние) для него. Давайте создадим для эксперимента простой счетчик

```
class CounterWidget extends StatefulWidget {  
  @override  
  _CounterWidgetState createState() => _CounterWidgetState();  
}
```

В классе виджета CounterWidget мы переопределяем метод createState, возвращая инстанс состояния в нем. А в классе состояния будет находиться вся бизнес-логика:

```
class _CounterWidgetState extends State<CounterWidget> {
    int _count = 42; // начальное значение

    @override
    Widget build(BuildContext context) {
        return Container(
            decoration: BoxDecoration(
                borderRadius: BorderRadius.all(Radius.circular(8.0)),
                color: Colors.amber[600],
            ),
            child: Row( // горизонтальное выравнивание
                mainAxisAlignment: MainAxisAlignment.min,
                mainAxisSize: MainAxisSize.center,
                crossAxisAlignment: CrossAxisAlignment.center,
                children: [
                    IconButton(
                        onPressed: () {
                            _decrement();
                        },
                        icon: Icon(Icons.remove)),
                    Text('$_count', style: TextStyle(fontSize: 20.0)),
                    IconButton(
                        onPressed: () {
                            _increment();
                        },
                        icon: Icon(Icons.add)),
                ],
            );
    }

    void _increment() {
        setState(() {
            _count++;
        });
    }
    void _decrement() {
        setState(() {
            _count--;
        });
    }
}
```

Как вы помните из предыдущей лекции, нижнее подчёркивание перед началом переменной говорит о том, что она приватна. Вот и в этом случае, мы делаем состояние приватным для виджета, чтобы только этот виджет мог его инстанцировать.

Состояние виджета хранится в объекте класса State. В котором нужно переопределить метод **build ()**. Когда нужно изменить состояние виджета, нам нужно вызывать **setState ()**, говоря таким образом ему: «Перерисуйся с новыми данными». Мы дергаем этот метод в функциях **_increment** и **_decrement**, в лямбде увеличивая и уменьшая значение переменной соответственно. Вот и все. Затем в методе build виджета, то есть в момент отрисовки, мы показываем величину счетчика в текстовом поле

```
Text ('$_count', style: TextStyle (fontSize: 20.0)),
```

Что интересного здесь по UI части?

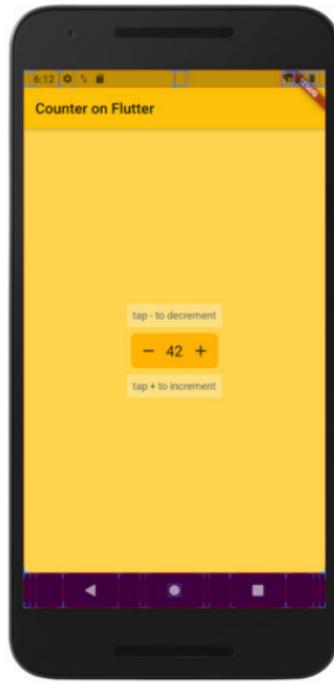
Кнопки для увеличения и уменьшения значения представлены виджетом **GestureDetector** – в своем конструкторе он получает функцию **onTap: ()** То есть нажмите. В ней в лямбде мы прописываем вызов функций **_increment** и **_decrement**.

Далее обратите внимание на то, как расположены виджеты кнопок и самого счетчика – они сгруппированы в **Row**. **Row** – это строка, то есть это массив виджетов: выстроенных по горизонтали. С помощью таких параметров, как **mainAxisSize**, **mainAxisAlignment**, **crossAxisAlignment** мы задаем выравнивание.

Антагонист виджета **Row** – это виджет **Column**. Он, как несложно догадаться, дает нам возможность выстраи-

вать виджеты по вертикали. Да, если вы ранее писали под Android, сразу скажу, что у нас нет виджета RelativeLayout, и LinearLayout, вместо них надо использовать Row и Column.

Что же, давайте запустим и проверим.



Кастомный виджет-счетчик

Мы добавили HintLabel-ы, а также использовали Column, так что код App стал выглядеть следующим образом:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Counter',
      theme: ThemeData(
        primarySwatch: Colors.amber,
      ),
      home: Scaffold(
        backgroundColor: Colors.amber[300],
        appBar: AppBar(
          title: Text('Counter on Flutter'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.min,
            mainAxisSize: MainAxisSize.min,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
              HintLabel('tap - to decrement'),
              SizedBox(height: 8.0),
              CounterWidget(),
              SizedBox(height: 8.0),
              HintLabel('tap + to increment')
            ],
          ),
        );
    );
}
```

Типы состояний: Ephemeral и App

Выделяют два типа состояний: Ephemeral и Application. Проще понять их, если рассматривать как область видимости – один для локального уровня, другой – для глобального.

Ephemeral state – это состояние с областью видимости виджета. Примеры: текущий прогресс слайдбара, текущая выбранная страница PageView, состояние checked у checkbox-а. В примере выше со счетчиком мы как раз пользуемся Ephemeral state – значение count мы храним в классе CounterWidgetState и только он знает о нем.

Application state (Inherited Widget). Используется, когда необходимо передавать данные между экранами, виджетами или пользовательскими сессиями. В таком типе состояний можно хранить, например, данные бизнес-логики, настройки, загруженные с сервера данные, пользовательскую корзину интернет-магазина и т. п. Для такого типа данных нет строго правила по методологии хранения: можно использовать setState или Redux, как вам будет угодно. Самый яркий пример использования App State – это Inherited Widget. В нем можно хранить данные, обращаясь к нему из любого виджета ниже по иерархии в приложении. В следующем уроке мы создадим приложение со списком виджетов.

Урок 4. Создание списка элементов

В этой главе:

- *ListView*
- *Создание списка через конструктор*
- *Создание списка с помощью Builder*
- *Добавляем заголовки*

ListView

В курсе Быстрый Старт мы создадим приложение, которое будет отображать погоду. В этом уроке мы создадим список с прогнозом погоды для конкретного города. Пока не будемходить в сеть, это будет в следующих уроках.

Чтобы создать список элементов во Flutter, нужно использовать виджет **ListView**

Есть 4 способа создания его:

- С помощью конструктора с указанием списка виджетов `ListView(List <Widget> widgets)` – используйте этот способ только в случае с простыми, небольшими статичными списками, поскольку `ListView` будет производить вычисления для каждого элемента из списка, а не только для тех, которые указаны на экране.
- С помощью билдера `ListView.builder` – подходит для

большого (бесконечного) количества элементов, поскольку builder вызывается только для тех элементов, которые видны на экране. На вход билдер принимает количество элементов и callback, с помощью которого надо отрисовать виджет по индексом i.

– Listview.separated – для списков с разделителями между элементами. Подходит для списков с фиксированным количеством элементов.

– ListView.custom – позволяет создать кастомный список.

Для начала создадим простой пример – по первому пункту.

Создание списка через конструктор

Раз мы решили создать список с прогнозом погоды, значит, нам нужны как минимум 2 класса – один для модели данных, и один для виджета с погодой. Создайте отдельный dart файл в той же папке lib и добавьте в него для начала класс модели:

```
class Weather {  
    static const String weatherURL = "http://openweathermap.org/img/w/";  
  
    DateTime dateTime;  
    num degree;  
    int clouds;  
    String iconURL;  
  
    String getIconUrl() {  
        return weatherURL + iconURL + ".png";  
    }  
  
    Weather(this.dateTime, this.degree, this.clouds, this.iconURL);  
}
```

Иконку облаков мы будем грузить с помощью удобного и простого метода *Image.network*

```
Image.network(weather.getIconUrl())
```

А формируем урл мы в методе *getIconUrl*

В этом же файле добавим класс виджета:

```
class WeatherListItem extends StatelessWidget {
    static var _dateFormat = DateFormat('yyyy-MM-dd - hh:mm');

    final Weather weather;

    WeatherListItem(this.weather);

    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: const EdgeInsets.all(16.0),
            child: Row(children: [
                Expanded(
                    flex: 3,
                    child: Text(_dateFormat.format(weather.dateTime))),
                Expanded(
                    flex: 1,
                    child: Text(weather.degree.toString() + " °C"),
                ),
                Expanded(
                    flex: 1,
                    child: Image.network(weather.getIconUrl())
                )));
    }
}
```

Expanded позволяет выровнять элементы в строке. Параметр flex задает приоритетацию, и работает примерно, как *weight* в LinearLayout.

Что еще здесь важного? Обратите внимание на переменную *_dateFormat*.

С помощью нее мы преображаем дату в читаемую строку, ее тип – DateFormat, который описан в пакете intl. Чтобы использовать, его нужно импортировать. Как это делается?

Импорт пакетов

Для начала нужно добавить в файл **pubspec.yaml** в разделе **dependencies** название пакета и версию следующим образом:

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  # The following adds the Cupertino Icons font to your application.  
  # Use with the CupertinoIcons class for iOS style icons.  
  cupertino_icons: ^0.1.2  
  
  intl: 0.15.8
```

Далее в файле, где планируется использовать этот пакет, импортировать его

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

Все, код компилируется.

Теперь используем написанный виджет и модель с данными. Для этого создадим отдельную страницу приложения и зададим для нее состояние.

```

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return WeatherForecastPage("Moscow");
  }
}

class WeatherForecastPage extends StatefulWidget {
  WeatherForecastPage(this.cityName);

  final String cityName;

  @override
  State<StatefulWidget> createState() {
    return _WeatherForecastPageState();
  }
}

class _WeatherForecastPageState extends State<WeatherForecastPage> {
  List<Weather> weatherForecast = [
    Weather(DateTime.now(), 20, 90, "04d"),
    Weather(DateTime.now().add(Duration(hours: 3)), 23, 50, "03d"),
    Weather(DateTime.now().add(Duration(hours: 6)), 25, 50, "02d"),
    Weather(DateTime.now().add(Duration(hours: 9)), 28, 50, "01d")
  ];
}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'ListView sample',
    theme: ThemeData(
      primarySwatch: Colors.amber,
    ),
    home: Scaffold(
      appBar: AppBar(
        title: Text('Weather forecast'),
      ),
      body: ListView( // здесь мы передаем список с элементами в
конструктор
      children: weatherForecast.map((Weather weather) {
        return WeatherListItem(weather);
      }).toList()),
    );
}
}

```

В классе состояния *WeatherForecastPageState* мы инициализируем список с объектами погоды в переменной *weatherForecast*. Прогноз представляет собой температуру и облачность на каждые три часа. Затем маппим этот спи-

сок в виджеты с помощью оператора *weatherForecast.map*. Созданные виджеты передаются в качестве параметра конструктору ListView. Готово

Создание списка с помощью Builder-а

Как мы помним, первый вариант создания списка подходит только для небольшого количества элементов. А если мы захотим отобразить погоду на недели вперед? Попробуем второй способ. Замените в методе build виджет body на следующий:

```
body: ListView.builder(  
    itemCount: weatherForecast.length,  
    itemBuilder: (BuildContext context, int index){  
        return WeatherListItem(weatherForecast[index]);  
    }),
```

Все! Что мы здесь видим? Вместо того, чтобы указывать весь список сразу, мы в builder передаем общее число элементов, а также *callback* на отрисовку i-элемента по его индексу. Так, ListView будет отрисовывать только те элементы, которые видны на экране.

Если мы посмотрим исходники этого конструктора, то увидим там, что колбэк является обязательным параметром.

@required IndexedWidgetBuilder itemBuilder,

Сам колбэк, а точнее его сигнатура возвращают Widget, а на вход принимают контекст и индекс.

```
/// Signature for a function that creates a widget for a given index,  
e.g., in a  
/// list.  
///  
/// Used by [ListView.builder] and other APIs that use lazily-  
generated widgets.  
///  
/// See also:  
/// * [WidgetBuilder], which is similar but only takes a  
[BuildContext].  
/// * [TransitionBuilder], which is similar but also takes a child.  
typedef IndexedWidgetBuilder = Widget Function(BuildContext context,  
int index);
```

Второй способ следует использовать в случае с динамическими списками элементов – когда вы точно заранее не можете сказать сколько будет элементов. Например, когда данные приходят с сервера.

В случае, например, с какими-то локальными ресурсами, когда нужно показать несколько элементов с иконками, рекомендуется использовать первый подход.

Заголовки в списке

Давайте добавим названия дней перед элементами списка погоды.

Для этого создадим абстрактный класс

```
abstract class ListItem {}
```

от него отнаследуем созданный ранее Weather. Дополнительно создадим класс заголовка. Он будет простой и будет содержать только дату.

```
class DayHeading extends ListItem {  
    final DateTime dateTime;  
  
    DayHeading(DateTime dateTime);  
}
```

Так же нам понадобится класс виджета для этого заголовка:

```
class HeadingListItem extends StatelessWidget implements  
ListWidgetItem {  
    static var _dateFormatWeekDay = DateFormat('EEEE');  
    final DayHeading dayHeading;  
  
    HeadingListItem(this.dayHeading);  
  
    @override  
    Widget build(BuildContext context) {  
        return ListTile(  
            title: Column(children: [  
                Text(  
                    "${_dateFormatWeekDay.format(dayHeading.dateTime)}  
${dayHeading.dateTime.day}.${dayHeading.dateTime.month}",  
                    style: Theme.of(context).textTheme.headline,  
                ),  
                Divider(),  
            ]),  
        );  
    }  
}
```

Теперь поменяем немного код инициализации списка с погодой. В цикле будем сравнивать item-ы погоды и даты, и, в случае, когда меняется день, будем добавлять заголовок:

```

List<ListItem> weatherForecast = List<ListItem>();

@Override
void initState() {
    var itCurrentDay = DateTime.now();
    weatherForecast.add(DayHeading(itCurrentDay)); // first heading
    List<ListItem> weatherData = [
        Weather(itCurrentDay, 20, "04d"),
        Weather(itCurrentDay.add(Duration(hours: 3)), 23, 50, "03d"),
        Weather(itCurrentDay.add(Duration(hours: 6)), 25, 50, "02d"),
        // ... добавляем прогнозы ручками, см. репозиторий
        Weather(itCurrentDay.add(Duration(hours: 42)), 24, 60, "02d"),
        Weather(itCurrentDay.add(Duration(hours: 45)), 20, 60, "02d")
    ];
    var itNextDay = DateTime.now().add(Duration(days: 1));
    itNextDay = DateTime(
        itNextDay.year, itNextDay.month, itNextDay.day, 0, 0, 0, 0);
    var iterator = weatherData.iterator;
    while (iterator.moveNext()) {
        var weatherDateTime = iterator.current as Weather;
        if (weatherDateTime.dateTime.isAfter(itNextDay)) {
            itCurrentDay = itNextDay;
            itNextDay = itCurrentDay.add(Duration(days: 1));
            itNextDay = DateTime(
                itNextDay.year, itNextDay.month, itNextDay.day, 0, 0, 0, 0,
                1);
            weatherForecast.add(DayHeading(itCurrentDay)); // next heading
        } else {
            weatherForecast.add(iterator.current);
        }
    }
    super.initState();
}

```

А теперь, внимание, следим за руками. В колбэке билдера ListView мы добавляем if – проверку на тип данных, и в зависимости от него рисуем либо заголовок, либо виджет погоды.

```

body: ListView.builder(
itemCount: weatherForecast.length,
itemBuilder: (BuildContext context, int index) {
    final item = weatherForecast[index];
    if (item is Weather) {
        return WeatherListItem(item);
    } else if (item is DayHeading) {
        return HeadingListItem(item);
    } else
        throw Exception("wrong type");
})));

```

В общем, все должно быть вам знакомо, если вы писали

adapter-ы с разными типами View в Android.

Урок 5. Загрузка данных с сервера

В этой главе:

Любой виджет в иерархии приложения будет иметь к нему доступ.

Если пока вы ничего не поняли, это ОК, сейчас мы посмотрим на примере, и все станет ясно.

Добавляем InheritedWidget в приложение погоды

- Асинхронность во Flutter
- Делаем запрос на сервер
- Парсим JSON
- Показываем полученные данные

Асинхронность во Flutter

Такие операции, как запрос на сервер, запись данных в базу, или определение геопозиции, могут занимать продолжительное время. Для того чтобы не блокировать UI долгой синхронной операцией, мы используем асинхронные. В Dart они представлены объектами класса Future. Объект Future представляет собой результат асинхронной операции, и может находиться в двух состояниях: выполнено и не выполнено. При вызове асинхронной функции мы получаем объект Future в невыполненнном состоянии.

Тип данных, возвращаемых Future объектом, указывается в дженерике как Future <T>. Если возвращать значение не нужно, можно указать тип void: Future <void>

Функцию надо пометить ключевым словом **async**, чтобы она выполняла работу асинхронно. Причем она будет выполняться синхронно, пока не встретит у себя слово **await**.

Посмотрим на примере:

```
Future<Response> performRequest() async {
    var uri = Uri.https(Constants.BASE_URL, Constants.API_URL);
    return await http.get(uri);
}
```

Функция **performRequest ()** выполняет запрос на сервер

и результат возвращается во future объекте.

Как теперь использовать данные, полученные с сервера, они ведь завернуты в объект Future. Очень просто. Нам нужно вызвать метод **then** на объекте Future:

```
void _loadData() {
    _isLoading = true;
    var httpResponseFuture = _performRequest();
    httpResponseFuture.then((response) {
        _isLoading = false; // здесь запрос уже выполнен и мы работаем с
    результатом
        if (response.statusCode == 200) {
            // Если все ок, парсим json ответа
        }else{
            // ошибка
        }
    });
}
```

Делаем запрос на сервер

Выше вы уже увидели, как выглядит асинхронный запрос на сервер. Давайте же теперь напишем его сами. Для примера мы будем использовать openweathermap API, и в конце урока у нас будет приложение с прогнозом погоды.

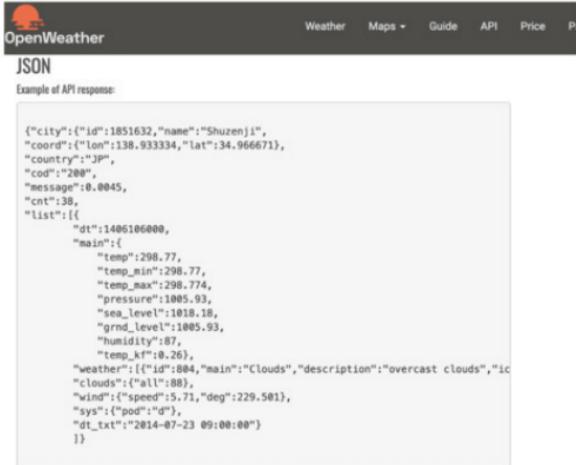
Во-первых, зарегистрируйтесь на сайте openweathermap.org для того, чтобы получить API Key.

Примечание: В России регистрация на сайте может быть недоступна, рассмотрите возможность использования TOR или VPN.

После регистрации и получения API Key, создайте в проекте файл констант, и добавьте в него следующие строки:

```
class Constants {  
    static const String WEATHER_BASE_SCHEME = 'https://';  
    static const String WEATHER_BASE_URL = 'api.openweathermap.org';  
    static const String WEATHER_IMAGES_PATH = "/img/w/";  
    static const String WEATHER_IMAGES_URL =  
        WEATHER_BASE_SCHEME + WEATHER_BASE_URL + WEATHER_IMAGES_PATH;  
    static const String WEATHER_FORECAST_URL = "/data/2.5/forecast";  
    static const String WEATHER_APP_ID = "<ваш API ключ>";  
}
```

Мы будем использовать почасовой (каждые 3 часа) прогноз по геокоординатам. Подробная документация находится [здесь](#). Откройте ссылку и скопируйте пример JSON ответа сервера в буфер обмена.



The screenshot shows a web browser displaying the OpenWeather API documentation. At the top, there's a navigation bar with links for Weather, Maps, Guide, API, Price, and Premium. Below the navigation, the word "JSON" is highlighted. Underneath, it says "Example of API response:" followed by a large code block. The code is a JSON object representing weather data for Shuzenji, Japan, at a specific time. It includes details like coordinates, weather conditions, temperature, pressure, humidity, wind speed, and a timestamp.

```
{"city": {"id": 1851632, "name": "Shuzenji", "coord": {"lon": 138.933334, "lat": 34.966671}, "country": "JP", "cod": "200", "message": "0.0045", "cnt": 38}, "list": [{"dt": 1406106000, "main": {"temp": 298.77, "temp_in": 298.77, "temp_max": 298.774, "pressure": 1005.93, "sea_level": 1018.18, "grnd_level": 1005.93, "humidity": 87, "temp_kf": 10.26}, "weather": [{"id": 804, "main": "Clouds", "description": "overcast clouds", "icon": "04d"}], "clouds": {"all": 88}, "wind": {"speed": 15.71, "deg": 229.501}, "sys": {"pod": "d"}, "dt_txt": "2014-07-23 09:00:00"}]}
```

Устанавливаем плагин в студии для конвертации JSON в Dart

Теперь нам нужно сделать маппинга JSON-а. То есть создать классы, представляющие данные от сервера. Сразу скажу, что Flutter не умеет Gson, и нужно создавать соответствующие данным классы и в них прописывать маппинг. Писать все это ручками было бы утомительно и неэффективно, поэтому мы воспользуемся плагином для IDEA, который сделает это за нас. Вы можете выбрать любой на выбор. Зайдите в настройки IDE, раздел Plugins, и в поиске вбейте flutter json. Я выбрал [FlutterJsonBeanFactory](#)

После установки перезапустите IDE.

Создайте пакет model – в него положим созданные на ос-

нове JSON классы

Кликните правой кнопкой мыши по пакету и выберите New-> JSONtoDartBeanAction

Скопируйте с <https://openweathermap.org/forecast5#geo5> JSON в оконце и жмите OK. Наши классы, автоматически сгенерированные, готовы для получения информации от сервера.

The screenshot shows the Android Studio project structure on the left and the generated Dart code on the right. The file `forecast_response.dart` is highlighted with a red rectangle.

```
22 class ForecastResponse {
23     String cod;
24     String message;
25     int cnt;
26     List<WeatherListBean> list;
27     CityBean city;
28
29     static ForecastResponse fromMap(Map<String, dynamic> map) {
30         if (map == null) return null;
31         ForecastResponse forecastResponseBean = ForecastResponse();
32         forecastResponseBean.cod = map['cod'];
33         forecastResponseBean.message = map['message'].toString();
34         forecastResponseBean.cnt = map['cnt'];
35         forecastResponseBean.list = List();
36         ..addAll(
37             (map['list'] as List ?? []).map((o) => WeatherListBean.fromMap(o)));
38         forecastResponseBean.city = CityBean.fromMap(map['city']);
39         return forecastResponseBean;
40     }
41
42     Map toJson() => {
43         "cod": cod,
44         "message": message,
45         "cnt": cnt,
46         "list": list,
47         "city": city,
48     };
49 }
50
51 //id : 1907296
52 //name : "Томарано"
53 //coord : {"lat":35.0164,"lon":139.0077}
54 //country : "none"
```

Http запросы

Для выполнения HTTP запросов во Flutter приложениях нужно использовать [пакет http](https://pub.dev/packages/http) <https://pub.dev/packages/http>. Обновите ваш **pubspec.yaml**, добавив в него зависимость:

dependencies:

http: <номер версии>

После выполните команду `get packages`, и добавьте в `main.dart` импорт пакета.

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

Простой GET запрос будет выглядеть следующим образом:

```
Future<http.Response> fetchData() {  
    return http.get('<API URL>');  
}
```

В нашем примере мы будем пользоваться несколько иной конструкцией, поскольку нам нужно будет делать параметризованный запрос. Мы воспользуемся [конструктором](#)

Uri.https

И выглядеть наша функция запроса погоды будет следующим образом:

```
Future<List<ListItem>> getWeather(double lat, double lng) async {
    var queryParameters = { // подготавливаем параметры запроса
        'APPID': Constants.WEATHER_APP_ID,
        'units': 'metric',
        'lat': lat.toString(),
        'lon': lng.toString(),
    };
    var uri = Uri.https(Constants.WEATHER_BASE_URL,
        Constants.WEATHER_FORECAST_URL, queryParameters);
    var response = await http.get(uri); // выполняем запрос и ждем
    // результата
    if (response.statusCode == 200) {
        var forecastResponse =
            ForecastResponse.fromJson(json.decode(response.body));
        if (forecastResponse.cod == "200") {
            // в случае успешного ответа парсим JSON и возвращаем список с
            // прогнозом
            return forecastResponse.list;
        } else {
            // в случае ошибки показываем ошибку
            Scaffold.of(context).showSnackBar(SnackBar(
                content: Text("Error ${forecastResponse.cod}"),
            ));
        }
    } else {
        // в случае ошибки показываем ошибку
        Scaffold.of(context).showSnackBar(SnackBar(
            content: Text("Error occurred while loading data from server"),
        ));
    }
    return List<ListItem>();
}
```

Как вы помните, мы используем координаты текущего места для получения погоды от API. И, соответственно, на вход функции `getWeather` передаем широту и долготу. Где же их взять?

Получение геопозиции

Для получения текущего местоположения устройства подключите пакет **geolocator**

dependencies:
geolocator: 5.1.3

Он позволяет получить как координаты, так и название текущего местоположения устройства. После обновления pubspec.yaml добавьте импорт пакета

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

Затем добавьте следующую функцию в класс с виджетом погоды:

```
Future<Placemark> getLocation() async {
    Geolocator geolocator = Geolocator()..forceAndroidLocationManager =
        true;
    Position position = await geolocator
        .getCurrentPosition(desiredAccuracy: LocationAccuracy.low); // получаем геопозицию
    List<Placemark> placemark = await Geolocator()
        .placemarkFromCoordinates(position.latitude,
        position.longitude); // определяем название места по геопозиции
    if (placemark.isNotEmpty) {
        return placemark[0]; // возвращаем первый элемент из списка
        полученных вариантов
    }
    return null;
}
```

Осталось только вызвать поочередно функцию определения координат, а затем функцию получения погоды. Выглядеть это будет следующим образом:

```
void _loadData() {
    _isLoading = true;
    var locationFuture = getLocation(); // получаем future на геопозицию
    locationFuture.then((placemark) { // берем value из результата future
        var weatherFuture =
            getWeather(placemark.position.latitude,
            placemark.position.longitude); // делаем запрос на получение погоды
        weatherFuture.then((weatherData) { // берем value response из
            future погоды
                initWeatherWithData(weatherData, placemark);
                _isLoading = false;
            });
        });
    });
}
```

Все, приложение умеет делать запрос, получать и парсить данные. Осталось сделать некоторые изменения в UI, чтобы созданные на прошлом уроке виджеты могли показать настоящую погоду.

Показываем полученные с сервера данные

Для того, чтобы приложение выглядело более user-friendly, добавим виджет прогресса на время загрузки данных с сервера. Для этого добавим следующие функции:

```
Widget get _pageToDisplay {
    if (_isLoading) {
        return _loadingView;
    } else {
        return _contentView;
    }
}

Widget get _loadingView {
    return Center(
        child: CircularProgressIndicator(), // виджет прогресса
    );
}
```

В зависимости от состояния загрузки, которым мы управляем в функции `_loadData`, мы будем отражать или `CircularProgressIndicator` или собственно, сам контент, то есть список.

Вынесем в отдельную функцию код создания контента из `body` виджета:

```
Widget get _contentView {
    return ListView.builder(
        itemCount: _weatherForecast == null ? 0 :
        _weatherForecast.length,
        itemBuilder: (BuildContext context, int index) {
            final item = _weatherForecast[index];
            if (item is ListBean) {
                return WeatherListItem(item);
            } else if (item is DayHeading) {
                return HeadingListItem(item);
            } else
                throw Exception("wrong type");
        });
}
```

Метод `build` меняется. Из него мы убрали код создания `ListView`. А вместо этого добавим вызов функции `_pageToDisplay`

```
@override
Widget build(BuildContext context) {
    return MaterialApp(
        title: 'Weather report',
        theme: ThemeData(
            primarySwatch: Colors.amber,
        ),
        home: Scaffold(
            appBar: AppBar(
                title: Text(_placeTitle),
            ),
            body: _pageToDisplay));
}
```

Также нам надо поменять код отрисовки виджета элемента списка, чтобы он брал данные не из старых мок-классов, а из нашей модели – классов JSON маппинга.

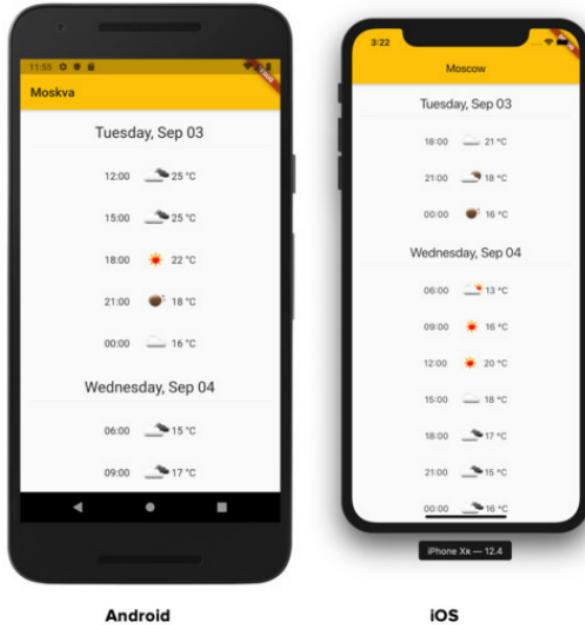
```
class WeatherListItem extends StatelessWidget implements
ListWidget {
    static var _dateFormatTime = DateFormat('HH:mm');

    final ListBean weather;

    WeatherListItem(this.weather);

    @override
    Widget build(BuildContext context) {
        return Padding(
            padding: const EdgeInsets.all(8.0),
            child: Row(mainAxisAlignment: MainAxisAlignment.center,
            children: [
                Padding(
                    padding: const EdgeInsets.only(right: 16.0),
                    child:
                    Text(_dateFormatTime.format(weather.getDateTime()),
                        style: Theme.of(context).textTheme.subhead)),
                Image.network(weather.getIconUrl(),
                    Text((weather.main.temp.toInt()).toString() + " \u00B0C",
                        style: Theme.of(context).textTheme.subhead)
                )));
    }
}
```

Можно запускать.



Android

iOS

Экран с прогнозом погоды на обеих платформах

Я не дизайнер, но получилось наглядно, и актуальную погоду показывает. Можно пользоваться.

Улучшим еще немного юзабилити, добавив PullToRefresh. Для этого во Flutter есть **RefreshIndicator**

В виджете контента корневым сделаем его:

```
return RefreshIndicator(  
    onRefresh: _onRefresh,  
    child: ListView.builder(
```

А в функцию обновления `onRefresh` добавим следующий код:

```
Future<Null> _onRefresh() async {
    Completer<Null> completer = Completer<Null>();
    var weatherFuture =
        _getWeather(_placemark); // делаем запрос на получение погоды
    weatherFuture.then((weatherData) {
        initWeatherWithData(weatherData);
        completer.complete(null);
    });
    return completer.future;
}
```

RefreshIndicator в качестве параметра принимает функцию обновления, которая должна возвращать Future объект. Именно это мы и делаем в функции: мы создаем объект специального типа **Completer**, который позволяет вернуть future по завершении обновления страницы. Также, Completer умеет возвращать ошибку. Принцип работы класса Completer хорошо описан в документации:

```
* If you do need to create a Future from scratch – for example,
* when you're converting a callback-based API into a Future-based
* one – you can use a Completer as follows:
```

То есть, по сути, Completer позволяет создавать Future объект.

Итак, мы сделали запрос на сервер, получили, распарсили

и показали пользователю полученную информацию на клиенте. То есть мы создали полноценное клиент-серверное приложение!

Все! На этом этапе уже можете считать себя Flutter-разработчиками.

Урок 6. Inherited Widgets, Elements, Keys

В этой главе:

- *Inherited Widget*
- *Elements*
- *Keys*

Inherited widgets

На третьем уроке мы выяснили, что виджеты во Flutter приложении образуют вложенную иерархичную структуру, начиная от самого верхнего уровня – App. Это удобно когда верстаешь экран, но крайне неудобно, если нам нужно передать данные от верхнего виджета к нижнему по всей иерархии, или запросить данные от нижнего с верхнего. Здесь нам на помощь приходит специальный инструмент, который разработчики из Google подготовили для нас – Inherited Widget.

Если раньше вы использовали такие конструкции, как:

Theme.of(context).textTheme

MediaQuery.of(context).size

то вы уже использовали InheritedWidget. Как и все

InheritedWidget они имеют специальный статичный метод of, который можно вызвать отовсюду вниз по иерархии дерева виджетов. Таким образом мы можем создавать центральное хранилище данных в приложении, или AppState.

Класс InheritedWidget наследуется от ProxyWidget, который наследует класс Widget.

В нем описан всего один метод

@protected

```
bool updateShouldNotify (covariant InheritedWidget  
oldWidget);
```

Это булев метод, который определяет, нужно ли обновлять дочерние виджеты. Предоставление данных дочерним виджетам организуется с помощью статичного метода of, который вызывает

```
BuildContext.inheritFromWidgetOfExactType ([type])
```

Схема использования InheritedWidget:

Сначала мы 1) создаем класс MyInheritedWidget, наследуя его от InheritedWidget, и добавляем в него нужную нам переменную. InheritedWidget иммутабельны, то есть нельзя изменить его состояние. Как же тогда обновлять переменную? Для этого нужно снова создать инстанс класса MyInheritedWidget.

А создавать инстанс мы будем в 2) специально созданном *Stateful виджете* – обертке *MyInheritedWidget*.

Всю логику по обновлению переменной будем класть в State этого виджета.

После создания этих классов нам остается только 3) поместить виджет-обертку в корень иерархии виджетов приложения. И затем 4) получать данные с помощью конструкции:

```
var myInheritedWidget = MyInheritedWidget.of(context);  
_variable = myInheritedWidget?.myVariable;
```

Любой виджет в иерархии приложения будет иметь к нему доступ.

Если пока вы ничего не поняли, это ОК, сейчас мы посмотрим на примере, и все станет ясно.

Добавляем *InheritedWidget* в приложение погоды

На прошлом уроке мы сумели получить геопозицию с помощью пакета *geolocator*, и затем передавали координаты в качестве параметров в сервис погоды. Однако, вы могли заметить, что данные о геопозиции сразу после получения передавались дальше по цепочке, нигде не сохраняясь. То есть, по сути, мы оперировали *EmpheralState*. А что, если мы захотим снова обратиться к сервису погоды, не обновляя местоположения? Ведь запрос на определение местоположения достаточно продолжительный. Для решения этого вопроса

нам нужно сохранить данные о местоположении на уровне приложения, то есть AppState. И тут нам как раз пригодится InheritedWidget.

Итак, выполним описанные выше 4 шага.

1) создадим класс для хранения местоположения

```
class LocationInfo extends InheritedWidget {  
    final Placemark placemark;  
  
    LocationInfo(this.placemark, Widget child)  
        : super(  
            child: child,  
        );  
  
    // статичный метод для получения инстанса класса  
    static LocationInfo of(BuildContext context) =>  
        context.inheritFromWidgetOfExactType();  
  
    @override  
    bool updateShouldNotify(LocationInfo oldLocationInfo) {  
        var oldLocationTime = oldLocationInfo  
            ?.placemark?.position?.timestamp?.millisecondsSinceEpoch  
        ??  
            0;  
        var newLocationTime =  
            placemark?.position?.timestamp?.millisecondsSinceEpoch ?? 0;  
  
        // сравниваем время в объектах местоположения, чтобы определить,  
        // нужно ли обновлять виджеты  
        if (oldLocationTime == 0 && newLocationTime == 0) {  
            // для случая первой загрузки  
            return true;  
        }  
        return oldLocationTime < newLocationTime;  
    }  
}
```

2) далее создадим стейт для виджета-обертки. В нем, переопределяем *initState*, чтобы вызвать метод *_loadData*, который будет отвечать за получение свежей геопозиции

```
class _LocationInheritedState extends State<LocationInheritedWidget> {
  // локальная переменная
  Placemark _placemark;

  // вся логика получения местоположения инкапсулируется в этом
  // виджете-обертке
  void _loadData() {
    var locationFuture = getLocation(); // получаем future на
    // геопозицию
    locationFuture.then((newPosition) {
      // берем value из результата future
      var placeFuture = getPlacemark(newPosition);
      placeFuture.then((newPlaceMark) {
        onPositionChange(newPlaceMark);
      });
    });
  };
}

@Override
void initState() {
  super.initState();
  _loadData();
}

void onPositionChange(Placemark newPlacemark) {
  setState(() {
    // обновляем локальную переменную
    // после чего произойдет вызов метода [build]
    _placemark = newPlacemark;
  });
}

@Override
Widget build(BuildContext context) {
  // в методе build происходит создание нового инстанса
  [LocationInfo]
  return LocationInfo(_placemark, widget.child);
}

Future<Position> getLocation() async {
  Geolocator geoLocator =
  Geolocator()..forceAndroidLocationManager = true;
  Position position = await geoLocator.getCurrentPosition(
    desiredAccuracy: LocationAccuracy.low); // получаем
  // геопозицию
  return position;
}
```

```
| Future<Placemark> getPlacemark(Position position) async {
|   List<Placemark> placemark = await
|     Geolocator().placemarkFromCoordinates(
|       position.latitude,
|       position.longitude); // определяем название места по
|     геопозиции
|     if (placemark.isNotEmpty) {
|       return placemark[
|         0]; // возвращаем первый элемент из списка полученных
|       вариантов
|     }
|     return null;
| }
```

3) Добавим *LocationInheritedWidget* в корень иерархии приложения. Обновленный main. dart будет выглядеть так

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';
import 'package:flutter_hello_world/LocationInfo.dart';
import 'package:flutter_hello_world/WeatherForecastPage.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return LocationInheritedWidget(
      child: WeatherForecastPage(),
    );
}
```

4) Теперь в *_WeatherForecastPageState* переопределим метод *didChangeDependencies*, который подписан на изменения в InheritedWidget-е, а также добавим переменную местоположения.

```
class _WeatherForecastPageState extends State<WeatherForecastPage> {
    // локальная переменная местоположения
    Placemark _placemark;

    bool _isLoading = false;

    Completer<void> _refreshCompleter;

    @override
    void initState() {
        super.initState();
        _refreshCompleter = Completer<void>();
        _loadData();
    }

    Future<void> _onRefresh() async {
        var weatherFuture =
            getWeather(_placemark.position.latitude,
            _placemark.position.longitude);
        weatherFuture.then((weatherForecast) {
            initWeatherWithData(weatherForecast, _placemark);
        });
        return _refreshCompleter.future;
    }
}
```

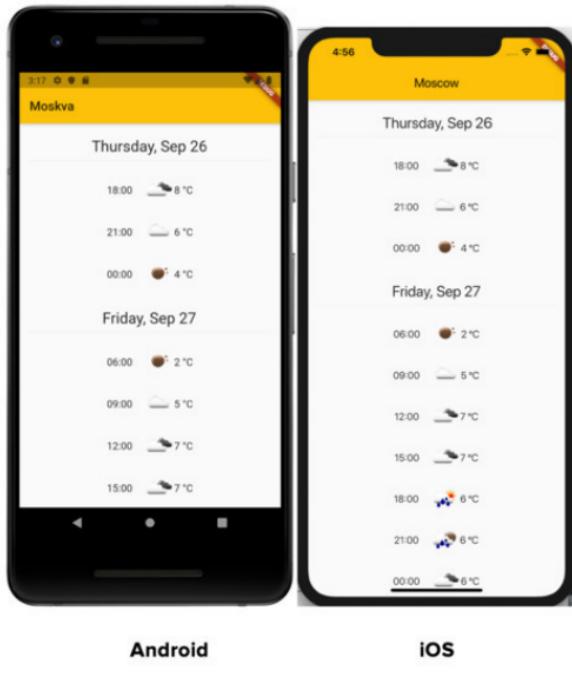
```
// метод вызывается, когда состояние объектов, от которых зависит
// этот виджет, меняется
@Override
void didChangeDependencies() {
    super.didChangeDependencies();

    //получем инстанс InheritedWidget-а
    var locationInfo = LocationInfo.of(context);
    //читаем оттуда местоположение
    _placemark = locationInfo?.placemark;
    // загружаем прогноз погоды
    _loadData();
}

void _loadData() {
    // если местоположения нет, то показываем progressBar
    _isLoading = true;
    if (_placemark == null) {
        return;
    }
    var weatherFuture = getWeather(_placemark?.position?.latitude,
        _placemark?.position?.longitude); // делаем запрос на
    // получение погоды
    weatherFuture.then((weatherData) {
        // берем value response из future погоды
        initWeatherWithData(weatherData, _placemark);
        _isLoading = false;
    });
} // осталное остается без изменений
}
```

* * *

Запускаем...



После рефакторинга внешне ничего не поменялось

С точки зрения пользователя все осталось таким же, а под капотом у нас теперь **InheritedWidget**.

Elements

Так что же, собственно, под капотом?

InheritedWidget содержит в себе список ссылок на все зависящие от него виджеты. Всякий раз, когда InheritedWidget пересоздается, и его данные меняются, он уведомляет по списку всех виджетов. Как это происходит? Заглянем в класс InheritedWidget. В нем мы видим строчку:

@override

InheritedElement createElement () => InheritedElement (this);

Это создается Element для InheritedWidget.

Сделаем здесь небольшое отступление и сначала разберемся с тем, что же такое Element.

Описание класса начинается так:

```
/// An instantiation of a [Widget] at a particular location in the
tree.
///
/// Widgets describe how to configure a subtree but the same widget
can be used
/// to configure multiple subtrees simultaneously because widgets are
immutable.
/// An [Element] represents the use of a widget to configure a
specific location
/// in the tree. Over time, the widget associated with a given element
can
/// change, for example, if the parent widget rebuilds and creates a
new widget
/// for this location.
```

То есть, UI-элементы, которые вы видите на экране есть

не что иное как объекты класса Element, которые описываются классом Widget. Именно Elements формируют дерево объектов на экране.

Жизненный цикл элемента:

- 1) Элемент создается методом `Widget.createElement`
- 2) Фреймворк вызывает метод **`mount`**, добавляя элемент в дерево на соответствующем месте. Начиная с этого момента, элемент становится активным (**active**) и может появиться на экране.
- 3) Далее у элемента можно вызывается метод **`update`** в случае, когда привязанный виджет изменяет свое состояние. В случае, если меняется `runtimeType` типа элемента (другой тип виджета) или ключ элемента, то на нем сначала нужно вызвать **`unmount`** и затем инфлейтить заново.
- 4) Для деактивации элемента его родитель может вызвать **`deactivateChild`**, переводя его состояние в **inactive**. Элемент в таком состоянии уже не видим на экране, и если его состояние не изменится, то в конце фрейма анимации на нем будет вызван **`unmount`**. Такой элемент становится **defunct** и уже не может быть возвращен в дерево.

Создание элементов – дорогостоящая операция и поэтому они должны быть по возможности переиспользованы. Это достигается с помощью ключей (Keys). С ключами разберемся дальше, а пока вернемся к InheritedElement и тому, как он уведомляет другие элементы, что надо перерисоваться.

Все просто, в нем описан метод, который пробега-

ет по всем зависимым элементам и вызывает функцию `notifyDependent`, которая в свою очередь, вызывает на виджете `didChangeDependencies()`

```
void notifyClients(InheritedWidget oldWidget) {
    assert(_debugCheckOwnerBuildTargetExists('notifyClients'));
    for (Element dependent in _dependents.keys) {
        assert(() {
            // check that it really is our descendant
            Element ancestor = dependent._parent;
            while (ancestor != this && ancestor != null)
                ancestor = ancestor._parent;
            return ancestor == this;
       }());
        // check that it really depends on us
        assert(dependent._dependencies.contains(this));
        notifyDependent(oldWidget, dependent);
    }
}

@protected
void notifyDependent(covariant InheritedWidget oldWidget, Element
dependent) {
    dependent.didChangeDependencies();
}
```

Keys

Ключи – это идентификаторы виджетов (Widget), элементов (Элементов), а также семантических узлов (SemanticNode).

Когда flutter перерисовывает UI, он проходит по всей иерархии элементов, сравнивая ТИП виджета и КЛЮЧ. Таким образом, если тип тот же, а ключ не задан, то перерисовывать будет нечего. Если ключ виджета, связанного с элементом, совпадает с ключом обновленного виджета, то элемент будет перерисован.

Ключи делятся на локальные и глобальные: LocalKey, GlobalKey. Локальные ключи должны быть уникальными среди элементов в пределах их родителя. Глобальные ключи должны быть уникальными в пределах всего приложения.

(!) В общем случае, **ключи не нужны Stateless-виджетам**, но нужны в случае со StatefulWidget, когда их несколько. Например, в списке.

Для того, чтобы показать работу ключей во Flutter, придется немного порефачить наш проект. Мы добавим экран со списком мест, для которых можно посмотреть погоду.

```
class PlacesPage extends StatefulWidget {
  PlacesPage({Key key}) : super(key: key);

  _PlacesPageState createState() => _PlacesPageState();
}

class _PlacesPageState extends State<PlacesPage> {

  // первоначальный тестовый список
  List<Placemark> _places = [
    Placemark(
      name: 'Moscow',
      country: 'Europe',
      administrativeArea: 'Moscow',
      position: Position(longitude: 37.6206, latitude: 55.7532)),
    Placemark(
      name: 'Paris',
      country: 'Europe',
      administrativeArea: 'Paris',
      position: Position(longitude: 2.2950, latitude: 48.8753)),
    Placemark(
      name: 'London',
      country: 'Europe',
      administrativeArea: 'London',
      position: Position(longitude: -0.1254, latitude: 51.5011)),
  ];
}
```

Далее в методе отрисовки обратите внимание на строку

```
key: Key(place.name),
```

у элемента **Dismissable**. Dismissible – это такой виджет списка, который можно удалять свайпом вправо или влево. При этом, чтобы ListView знал как ему корректно перерисовываться, ему нужно знать какой именно элемент удален. Мы задаем в качестве ключа название места, таким образом Flutter может их идентифицировать при перерисовке.

```
    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: Text("Places"),
        ),
        body: Column(children: <Widget>[
          Row(mainAxisAlignment: MainAxisAlignment.start, children:
<Widget>[
            Expanded(
              child: InkWell(
                onTap: () => _onItemTapped(null),
                child: Padding(
                  padding: const EdgeInsets.all(16.0),
                  child: Text("Current position",
                    textAlign: TextAlign.left,
                    style: new TextStyle(
                      fontSize: 16.0,
                      color: Colors.black,
                    )));
            ),
          ],
        ]),
        Divider(
          height: 4,
          thickness: 2,
        ),
      );
    }
  }
}
```

```
| Expanded(
|   child: ListView.builder(
|     itemCount: _places.length,
|     itemBuilder: (context, index) {
|       final place = _places[index];
|       return Dismissible(
|         key: Key(place.name),
|         onDismissed: (direction) {
|           setState(() {
|             _places.removeAt(index);
|           });
|           Scaffold.of(context)
|             .showSnackBar(SnackBar(content: Text("$place
removed")));
|         },
|         background: Container(
|           color: Colors.red,
|         ),
|         child: ListTile(
|           title: Text(_preparePlaceTitle(place)),
|           onTap: () => _onItemTapped(place),
|         ),
|       );
|     },
|   );
| },
| ***;
| ***;
| }
```

Что еще интересного мы видим здесь? Виджет **InkWell** – это просто прямоугольная область, которая реагирует на нажатия. Вот и все, собственно касательно ключей на сегодня. Тема довольно большая, поэтому дальнейшая работа со списком мест, который мы создали будет продолжена в следующей главе, где мы рассмотрим подключение пакетов Google Maps, а также TimeZone для добавления нового местоположения.

Урок 7. Навигация между экранами, Работа с Google Maps

В этой главе:

- Навигация с помощью *MaterialPageRoute*
- Интегрируем *Google Maps*
- Интегрируем пакет *timezone*

Навигация с помощью **MaterialPageRoute**

В конце прошлого урока мы создали экран со списком мест, но не сделали переход на экран погоды. Да и вообще, многие важные подробности, связанные с показом списка мест, их добавлением, были опущены. Сейчас мы ликвидируем эти пробелы.

Итак, приступим. Добавьте в класс *_PlacesPageState* следующие функции

```

/// генерирует название места на основе объекта [Placemark]
String _preparePlaceTitle(Placemark placemark) {
    var placeTitle = "";
    if (placemark.country != null) {
        placeTitle = placemark.country;
    }
    if (placemark.administrativeArea != null) {
        placeTitle = placeTitle + ", " + placemark.administrativeArea;
    } else if (placemark.name != null) {
        placeTitle = placeTitle + ", " + placemark.name;
    }
    return placeTitle;
}

/// Обработчик нажатия на элемент списка – переход на экран погоды
void _onItemTapped(Placemark place) {
    Navigator.push(
        context,
        MaterialPageRoute(builder: (context) =>
    WeatherForecastPage(place)),
    );
}

/// Обработчик нажатия на плавающую кнопку – добавление нового места
void _onAddNew() async {
    final result = await Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => MapPage()),
    ); // ждем добавленное место
    setState(() {
        if (result != null) {
            _places.add(result);
        }
    });
}

```

За навигацию в приложениях Flutter отвечает виджет под названием `Navigator`. Для осуществления операции смены экрана нужно вызвать метод `push` с параметром `MaterialPageRoute`, в конструкторе которого в лямбде метода `build` нужно передать тот виджет (экран), который мы хотим поместить поверх.

Обратите внимание, что функция `onAddNew` имеет модификатор `async`, а также `setState`. Таким образом, мы будем

ждать результат этого экрана – объект местоположения, который мы выберем на карте.

Теперь добавим экран с картой.

Интегрируем Google Maps

Примечание. Предполагается, что у вас уже есть com.google.android.geo. API_KEY. Он нужен для получения данных от серверов Google. Если у вас его еще нет, то сначала [получите его](https://developers.google.com/maps/documentation/android-sdk/get-api-key) согласно инструкции <https://developers.google.com/maps/documentation/android-sdk/get-api-key>

Добавьте в AndroidManifest. xml строки

```
<meta-data      android:name="com.google.android.geo.  
API_KEY» android: value="<!-- YOUR API KEY --&gt;"/&gt;</pre>
```

А для iOS в ios/Runner/AppDelegate

GMServices.provideAPISKey («YOUR API KEY»)

Далее в pubspec. yaml

google_maps_flutter: ^0.5.21+

И выполните команду *flutter packages get*

Теперь можно писать код. Создайте файл *map_page. dart* и добавьте в него:

```

import 'dart:collection';
import 'package:flutter/material.dart';
import 'package:flutter_hello_world/LocationInfo.dart';
import 'package:geolocator/geolocator.dart';
import 'package:google_maps_flutter/google_maps_flutter.dart';

class MapPage extends StatefulWidget {
  @override
  _MapPageState createState() => _MapPageState();
}

class _MapPageState extends State<MapPage> {
  GoogleMapController _mapController;
  Marker positionMarker;
  Set<Marker> _markers = HashSet<Marker>();
  Placemark _placemark;

  bool _isLoading = false;
  LatLng _markerPosition = LatLng(55.7532, 37.6206);

  void _onMapCreated(GoogleMapController controller) {
    _mapController = controller;
  }

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    _loadData();
  }
}

```

Отрисовка карты осуществляется виджетом **GoogleMap**

```

/// отрисовка карты осуществляется виджетом [GoogleMap]
Widget get _contentView {
  return GoogleMap(
    onMapCreated: _onMapCreated,
    markers: _markers,
    mapType: MapType.normal,
    initialCameraPosition: CameraPosition(
      target: _markerPosition,
      zoom: 11.0,
    ),
    onTap: (latLng) => _updatePlaceMark(latLng),
  );
}

```

При старте экран будет обращаться к местоположению, указанному в InheritedWidget и с помощью него рисовать виджет.

```

void _ loadData() {
    _isLoading = true;
    _initPlaceMark();
    if (_placemark != null) {
        setState(() {
            _markerPosition =
                LatLng(_placemark.position.latitude,
                    _placemark.position.longitude);
            reInitMarker();
            _isLoading = false;
        });
    }
}

/// читаем данные о местоположении из InheritedWidget [LocationInfo]
void _initPlaceMark() {
    if (_placemark == null || _placemark?.position == null) {
        //получим инстанс InheritedWidget-а
        var locationInfo = LocationInfo.of(context);
        //читаем оттуда местоположение
        _placemark = locationInfo?.placemark;
    }
}

```

Добавим также функцию, которая будет перерисовывать маркер на карте в случае клика на карте или перемещения маркера.

```

/// вызываем эту функцию при тапе на карте или перемещении маркера
/// обновляем координаты и данные местоположения для маркера
void _updatePlaceMark(LatLng latLng) {
    if (_isReinitingMarker) {
        return;
    }
    _isReinitingMarker = true;
    setState(() {
        _markerPosition = latLng;
    });
    var placeFuture = LocationHelper.getPlacemark(
        _markerPosition.latitude, _markerPosition.longitude);
    placeFuture.then((newPlaceMark) {
        setState(() {
            _placemark = newPlaceMark;
            reInitMarker();
        });
    });
}

```

Весь код экрана MapPage вы можете найти в репозитории

проекта.

Интегрируем пакет timezone

От сервиса погоды данные о дате приходят в формате UTC. Чтобы данные о времени для разных точек земного шара отображались корректно, нам потребуется знать timezone (часовой пояс) для определенного местоположения. В pub.dev/packages есть пакет [timezone](https://pub.dev/packages/timezone) (<https://pub.dev/packages/timezone>). В нем содержится вся база IANA (<https://www.iana.org/time-zones>) часовых поясов и соответствующих им координатам. Подключим эту библиотеку.

На странице readme присутствует краткая инструкция по подключению. Однако, она неполная и содержит не всю информацию. Более подробная инструкция есть на [Medium](https://medium.com/flutter-community/working-with-timezones-in-flutter-1c304089dcf9) (<https://medium.com/flutter-community/working-with-timezones-in-flutter-1c304089dcf9>)

Во первых, добавьте в **pubspec.yaml**

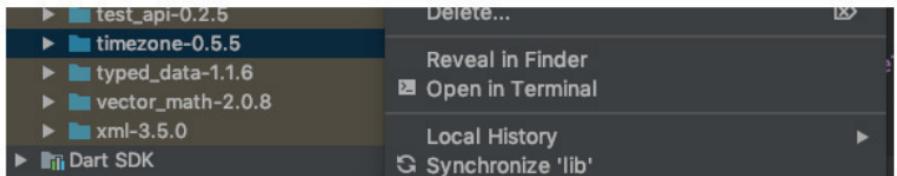
```
dependencies:  
  timezone: ^0.5.4
```

Затем выполните в терминале команду

```
flutter pub get
```

После этого вам нужно перейти в папку с установленными пакетами. В Android Studio в правой части IDE, в дереве

проекта раскройте **External Libraries-> Dart packages**, затем правой кнопкой жмем на пакете timezone, чтобы открыть папку в terminal.



Настройка пакета timezone

В окне терминала наберите **cd..** чтобы перейти на уровень выше и выполните команду

```
flutter pub run tool/get -s 2019b
```

После этого в `pubspec.yaml` укажите базу данных в качестве ресурса.

```
assets:  
  - packages/timezone/data/2019b.tzf
```

И последний шаг: инициализация библиотеки. Измените функцию `main` так, чтобы она выглядела следующим образом:

ЗОМ:

```
void main() async {
    WidgetsFlutterBinding.ensureInitialized();
    var byteData = await rootBundle
        .load('packages/timezone/data/2019b.tzf');
    initializeDatabase(byteData.buffer.asUint8List());
    runApp(MyApp());
}
```

Теперь можно пользоваться библиотекой. В нашем случае мы применим ее при показе элементов списка с прогнозом – каждое время нужно показывать с учетом часового пояса. В функции getWeather класса _WeatherForecastPageState мы будем парсить место placemark в таймзону и учитывать часовой пояс в элементе.

```
try {
    var locationZone =
"${_placemark.country}/${_placemark.administrativeArea.replaceAll(" ", "-")}";
    final placeTime =
TZDateTime.from(itCurrentDay, getLocation(locationZone));
    placeTimeZoneOffset = placeTime.timeZoneOffset;
} on Exception catch (e) {
    placeTimeZoneOffset = deviceTimeZoneOffset;
}
```

...

и далее сеттим оффсет таймзоны в элемент, который как вы помните, учитывает этот оффсет при показе времени:

```
while (iterator.moveToNext()) {
    var weatherItem = iterator.current as ListBean;
    weatherItem.timeZoneOffset = placeTimeZoneOffset;
    ...
    DateTime getDateTime() {
        var dateTime = DateTime.parse(dtTxt);
        if (timeZoneOffset != null) {
            return dateTime.add(timeZoneOffset); // учитываем часовой пояс
        } else {
            return dateTime;
        }
    }
}
```

Теперь время для какой-то отдельной точки будет отображаться с учетом часового пояса.

Урок 8. SQLite, Clean Architecture

В этой главе:

- Подключаем *SQLite*
- Реализуем паттерн *Repository* для списка мест
- Реализуем паттерн *Repository* для получения погоды

Подключаем SQLite

Мы добавили возможность добавлять и удалять места в списке мест. Это наглядно, но с точки зрения пользователя крайне неудобно то, что каждый раз при заходе в приложение список мест остается неизменным. Конечно, ведь он у нас никак не сохраняется, а инициализируется каждый раз заново при старте приложения (экрана). Сделаем как полагается, с сохранением в локальную базу.

Для Flutter существует реализация знакомой вам *SQLite* – **sqflite**

Добавьте ее в **pubspec.yaml** вместе с пакетом **path_provider** для работы с файловой системой

```
sqflite:  
  path_provider: ^1.4.0
```

После выполнения команды flutter packages get создадим файл **db_provider. dart**, а в нем – класс

```
class DBProvider {  
    DBProvider.();  
    static final DBProvider db = DBProvider._();  
    Database _database;  
}
```

DBProvider – это синглтон, который будет предоставлять нам возможность работать с базой данных. Мы будем выполнять команды, вызывая их на инстансе этого класса.

Метод *get database* осуществляет ленивую инициализацию объекта класса **Database** из package: sqflite/sqflite. dart

```
Future<Database> get database async {  
    if (_database != null) return _database;  
    // if _database is null we instantiate it  
    _database = await initDB();  
    return _database;  
}  
  
initDB() async {  
    Directory documentsDirectory = await  
        getApplicationDocumentsDirectory();  
    String path = join(documentsDirectory.path, DB_NAME);  
    return await openDatabase(path, version: 1, onOpen: (db) {},  
        onCreate: (Database db, int version) async {  
            await db.execute(CREATE_PLACES_TABLE);  
        });  
}
```

Теперь добавим использованные для SQL строковые константы

```
static const String DB_NAME = "flutter_weather.db";  
static const String PLACES_TABLE_NAME = "Places";  
static const String CREATE_PLACES_TABLE = "CREATE TABLE  
$PLACES_TABLE_NAME ("  
    "id INTEGER PRIMARY KEY,"  
    "name TEXT,"  
    "isoCountryCode TEXT,"  
    "country TEXT,"  
    "postalCode TEXT,"  
    "administrativeArea TEXT,"  
    "subAdministrativeArea TEXT,"  
    "locality TEXT,"  
    "longitude REAL,"  
    "latitude REAL,"  
    "timestamp INTEGER"  
");
```

Для того, чтобы мы могли идентифицировать объекты мест (Placemark), нам нужно задать каждому из них id. Но, поскольку класс Placemark описан в пакете geolocator-а, напишем свою обертку для локального хранения мест:

```
class PlacemarkLocal {
    int id;
    Placemark placemark;

    PlacemarkLocal({
        this.id,
        this.placemark,
    });

    factory PlacemarkLocal.fromMap(Map<String, dynamic> mapStr) =>
        new PlacemarkLocal(
            id: mapStr["id"],
            placemark: Placemark(
                name: mapStr["name"],
                isoCountryCode: mapStr["isoCountryCode"],
                country: mapStr["country"],
                administrativeArea: mapStr["administrativeArea"],
                subAdministrativeArea: mapStr["subAdministrativeArea"],
                position: Position(
                    longitude: mapStr["longitude"],
                    latitude: mapStr["latitude"],
                    timestamp: mapStr["timestamp"],
                ),
            ),
        );
}

Map<String, dynamic> toMap() => {
    "id": id,
    "name": placemark.name,
    "isoCountryCode": placemark.isoCountryCode,
    "country": placemark.country,
    "administrativeArea": placemark.administrativeArea,
    "subAdministrativeArea": placemark.subAdministrativeArea,
    "longitude": placemark.position?.longitude,
    "latitude": placemark.position?.latitude,
    "timestamp":
        placemark.position?.timestamp?.millisecondsSinceEpoch,
};
}
```

И далее добавим методы для манипулирования данными

```
/// обновление места
updatePlacemark(PlacemarkLocal newPlacemark) async {
    final db = await database;
    var res = await db.update(PLACES_TABLE_NAME, newPlacemark.toMap(),
        where: "id = ?", whereArgs: [newPlacemark.id]);
    return res;
}

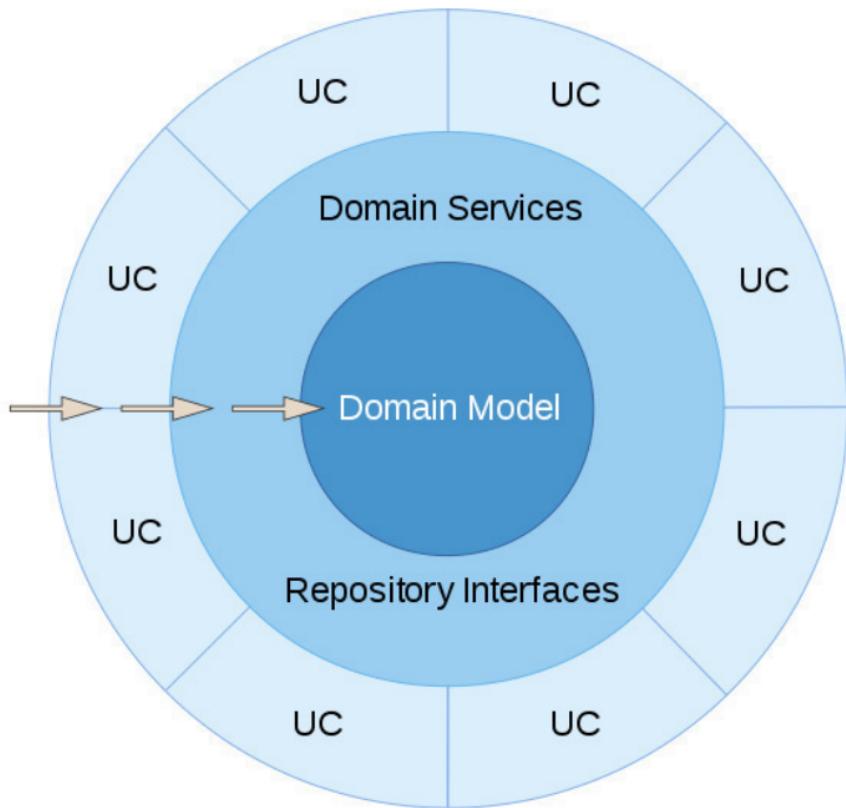
/// получение места
getPlacemark(int id) async {
    final db = await database;
    var res = await db.query("Client", where: "id = ?", whereArgs:
    [id]);
    return res.isNotEmpty ? PlacemarkLocal.fromMap(res.first) : null;
}

/// получение списка всех мест
Future<List<PlacemarkLocal>> getAllPlacemarks() async {
    final db = await database;
    var res = await db.query(PLACES_TABLE_NAME);
    List<PlacemarkLocal> list = res.isNotEmpty
        ? res.map((c) => PlacemarkLocal.fromMap(c)).toList()
        : [];
    return list;
}

/// удаление места
deletePlacemark(int id) async {
    final db = await database;
    return db.delete(PLACES_TABLE_NAME, where: "id = ?", whereArgs:
    [id]);
}
```

Сейчас мы, конечно, можем добавить в файле **places_page.dart** вызовы **DBProvider**. Однако, это будет не совсем корректно. Вдруг наш список мест станет приходить с сервера. Тогда придется переписывать код виджета. Что совсем неудобно, а с учетом сложной иерархичной структуры виджетов во flutter эта задача может стать сложной и вести к новым ошибкам. К счастью, существует надежное и проверенное решение – **clean architecture**.

Реализуем паттерн Repository для списка мест



Источник – commons.wikimedia.org

Если вы вдруг не знаете что такое clean architecture, обязательно изучите эту тему отдельно. Если вкратце, то у нас есть

несколько уровней: уровень домена данных (бизнес-логика, репозитории), уровень адаптеров интерфейсов (интеракторы, абстракции репозиториев) и уровень представления данных (UI, презентеры). Причем высшие слои зависят от низших, но ни в коем случае наоборот.

Доступ к данным мы сейчас выносим на уровень репозитория. А UI, то есть виджеты, будут обращаться к репозиторию посредством, например, презентера, или, как мы увидим дальше, с помощью специальных **BLoC**-ов.

Итак, создадим репозиторий для списка мест. Создайте package repository в папке lib. А в нем – файл **places_repository.dart**

```
class PlacesRepository {  
  PlacesRepository();  
}
```

Перенесем в него метод инициализации списка мест

```
List<Placemark> _initPlaces = [  
    Placemark(  
        name: 'Moscow',  
        country: 'Europe',  
        administrativeArea: 'Moscow',  
        position: Position(longitude: 37.6206, latitude: 55.7532)),  
    Placemark(  
        name: 'New York',  
        country: 'America',  
        administrativeArea: 'New York',  
        position: Position(longitude: -73.9739, latitude: 40.7715)),  
    Placemark(  
        name: 'Los Angeles',  
        country: 'America',  
        administrativeArea: 'Los_Angeles',  
        position: Position(longitude: -122.4663, latitude: 37.7705)),  
    Placemark(  
        name: 'Paris',  
        country: 'Europe',  
        administrativeArea: 'Paris',  
        position: Position(longitude: 2.2950, latitude: 48.8753)),  
    Placemark(  
        name: 'London',  
        country: 'Europe',  
        administrativeArea: 'London',  
        position: Position(longitude: -0.1254, latitude: 51.5011)),  
];
```

И добавим методы для работы оперирования с объектами мест посредством БД SQLite через подготовленный ранее DBProvider

```
Future<List<PlacemarkLocal>> getPlaces() async {
    var placesFuture = DBProvider.db.getAllPlacemarks();
    var placesList = await placesFuture;
    if (placesList.isEmpty) {
        await putPlaces(_initPlaces);
    }
    return await DBProvider.db.getAllPlacemarks();
}

addPlace(Placemark placemark) async {
    await DBProvider.db.addPlace(placemark);
}

updatePlacemark(PlacemarkLocal newPlacemark) async {
    await DBProvider.db.updatePlacemark(newPlacemark);
}

putPlaces(List<Placemark> places) async {
    for (var place in places) {
        await DBProvider.db.addPlace(place);
    }
}

deletePlacemark(int id) async {
    DBProvider.db.deletePlacemark(id);
}
```

И теперь мы можем добавлять в класс DBProvider методы, необходимые нам для выполнения операций над местами.

```
/// получение списка всех мест
Future<List<PlacemarkLocal>> getPlaces() async {
    var placesFuture = DBProvider.db.getAllPlacemarks();
    var placesList = await placesFuture;
    if (placesList.isEmpty) {
        putPlaces(_initPlaces);
    }
    return await DBProvider.db.getAllPlacemarks();
}

/// добавление места
addPlace(Placemark placemark) async {
    await DBProvider.db.addPlace(placemark);
}

/// обновление места
updatePlacemark(PlacemarkLocal newPlacemark) async {
    await DBProvider.db.updatePlacemark(newPlacemark);
}

/// добавление списка мест
putPlaces(List<Placemark> places) async {
    for (var place in places) {
        await DBProvider.db.addPlace(place);
    }
}

/// удаление места
deletePlacemark(int id) async {
    DBProvider.db.deletePlacemark(id);
}
```

Все, на этом репозиторий списка мест готов. То есть мы создали класс, который внутри себя сам подготавливает для внешнего слоя (UI) данные и выдает их. Чтобы лучше понять, создадим еще аналогичный репозиторий для получения данных о погоде.

Реализуем паттерн Repository для получения погоды

Добавьте файл **weather_repository.dart** с классом репозитория погоды.

```
class WeatherRepository {  
    final WeatherApiClient weatherApiClient;  
  
    WeatherRepository({@required this.weatherApiClient})  
        : assert(weatherApiClient != null);  
  
    Future<ForecastResponse> getWeather(Placemark placeMark) async {  
        return await weatherApiClient.fetchWeather(placeMark);  
    }  
}
```

Как вы видите, репозиторий просто-напросто инкапсулирует асинхронное получение данных о погоде. В нашем случае мы напрямую дергаем сервер с помощью объекта класса **WeatherApiClient**.

В случае, если бы мы хотели кэшировать данные, то эта логика была бы здесь, в репозитории.

WeatherApiClient отвечает за обращение к серверу, и в него мы переносим запрос вместе с **httpClient**-ом

```
class WeatherApiClient {
    final http.Client httpClient;

    WeatherApiClient({@required this.httpClient}) : assert(httpClient != null);

    Future<ForecastResponse> fetchWeather(Placemark placeMark) async {
        if (placeMark == null) {
            return null;
        }

        double lat = placeMark?.position?.latitude;
        double lng = placeMark?.position?.longitude;

        var queryParameters = {
            /// подготавливаем параметры запроса
            'APPID': Constants.WEATHER_APP_ID,
            'units': 'metric',
            'lat': lat.toString(),
            'lon': lng.toString(),
        };

        var uri = Uri.https(Constants.WEATHER_BASE_URL,
            Constants.WEATHER_FORECAST_URL, queryParameters);

        /// выполняем запрос и ждем результата
        var response = await http.get(uri);

        /// парсим JSON и возвращаем список с прогнозом
        var forecastResponse =
            ForecastResponse.fromMap(json.decode(response.body));

        return forecastResponse;
    }
}
```

На этом репозиторий готов, и мы можем получать данные. Однако, нам нужно еще их парсить и по необходимости маппить, то есть производить над ними некие операции, прежде, чем показать пользователю. Это мы будем делать в этом же слое с помощью специальных компонентов – **BloC**-ов

Урок 9. BLoC, Streams

В этой главе:

- *BLoC*
- *Streams*
- *Используем BLoC для примера со счетчиком*
 - *Рефакторим приложение погоды с использованием BLoC-библиотеки*

BLoC

Во всех примерах ранее мы меняли состояние виджета с помощью функции `setState`. На самом деле это достаточно удобно, когда `state` виджета имеет одну-две переменные и не содержит сложной логики. Однако, как вы наверняка знаете, приложения в реальном мире могут иметь достаточно сложную и запутанную бизнес логику, и если реализовывать всю ее в `State` классе виджета, то получится гигантский, трудноподдерживаемый спагетти-код. Для грамотной композиции и составления архитектуры кода такого проекта мы можем применить BLoC. Что же такое BLoC?

Буквально, это акроним от `Business Logic Component`. То есть компонент, который инкапсулирует в себе слой бизнес-логики приложения. Это тот же уровень, на котором работает презентер в MVP, а точнее ViewModel в MVVM.

Кстати, с ViewModel здесь крайне уместно провести аналогию, поскольку также, как и в MVVM, BLoC позволяет UI подписываться на изменения состояния. Вкратце, его можно представить простой диаграммой:



Схема работы паттерна BLoC

На вход поступают события, а на выходе имеем State, который используется Widget-ами.

Что же из себя представляет BLoC в коде? По сути, это просто класс с реактивными потоками внутри себя, которые, в свою очередь, манипулируют поступающими извне данными и позволяют UI подписываться на результат вычислений.

Здесь нужно остановиться и подробнее разобрать как в Dart представлено реактивное программирование и что такое Streams.

Streams

Streams – это потоки данных, испускаемых последовательно в количестве от 0 до n. По аналогии с RxJava, это Observable. Это чистая реактивщина, которая присутствует в Dart из коробки.

Для того, чтобы получать данные из потока Stream, нужно подписаться на него.

```
final _stream = SomeGenerator().stream;
final subscription = _stream.listen(
  (data) => print('$data'),
);
```

Важный момент: По умолчанию на Stream может подписаться только один слушатель. Чтобы один Stream могли слушать несколько подписчиков, нужно вызвать на stream-е метод **asBroadcastStream**

Также можно слушать на ошибки, то есть обрабатывать их с помощью соответствующих методов. Помимо этого, есть очень удобный флаг `cancelOnError`, который позволяет приложению не падать в случае ошибки в потоке. Что очень удобно, и не надо писать `onErrorReturn`, как в RxJava.

```
final subscription = _stream.listen(  
    (data) => print('$data'),  
),  
onError:(err){  
    print('err')  
},  
cancelOnError: false,  
onDone:(){  
    print('Done')  
}  
);
```

Так же как и в Rx, у Stream-ов есть оператор **map**, который позволяет фильтровать испускаемые элементы:

```
final subscription = _ SomeGenerator().stream  
.where((data) => data.isValid() == true)  
.map((data) => 'Valid $data')  
.listen(  
    (data) => print('$data'),  
);
```

StreamController

Для того, чтобы создать **Stream** вручную, нужно использовать класс **StreamController**. Это обертка над потоком Stream, которая позволяет отправлять данные в поток. Если мы посмотрим исходник класса, то в нем все достаточно просто. Stream с дженерик-типов и колбеки onListen, onPause, onResume, onCancel.

```
abstract class StreamController<T> implements StreamSink<T> {  
    /** The stream that this controller is controlling. */  
    Stream<T> get stream;  
  
    * * *  
  
    /**  
     * The callback which is called when the stream is listened to.  
     *  
     * May be set to 'null', in which case no callback will happen.  
     */  
    ControllerCallback get onListen;  
  
    void set onListen(void onListenHandler());  
  
    /**  
     * The callback which is called when the stream is paused.  
     *  
     * May be set to 'null', in which case no callback will happen.  
     *  
     * Pause related callbacks are not supported on broadcast stream  
     controllers.  
     */  
    ControllerCallback get onPause;  
  
    void set onPause(void onPauseHandler());  
  
    /**  
     * The callback which is called when the stream is resumed.  
     *  
     * May be set to 'null', in which case no callback will happen.  
     *  
     * Pause related callbacks are not supported on broadcast stream  
     controllers.  
     */  
    ControllerCallback get onResume;  
  
    void set onResume(void onResumeHandler());
```

Для того чтобы отослать событие/данные в поток Stream, нужно на StreamController вызвать метод **sink.add (<данные>)**. Как это работает на практике, мы увидим дальше в примере со счетчиком, а сейчас отметим важный момент как использовать Stream в UI.

Чтобы UI, то есть виджет, мог обновляться в соответствии с изменениями, поступающими от потока Stream, этот виджет нужно обернуть в специальный виджет **StreamBuilder**. Этот виджет имеет параметр stream, в котором мы указыва-

ем поток с данными, а также функцию билдера, в котором, используя получаемые от потока данные, можно отрисовывать виджет.

```
StreamBuilder(  
    stream: _myStream,  
    initialData: <объект>,  
    builder: (BuildContext context, AsyncSnapshot<Тип данных> snapshot)  
{  
    return Text('${snapshot.data}');  
},  
,
```

Используем BLoC для примера со счетчиком

А теперь, чтобы прочитанное лучше усвоилось, создадим BLoC с нуля для нашего простого приложения счетчика из третьей лекции.

Для начала создадим простые классы, которые будут представлять собой события инкремента и декремента

```
abstract class CounterEvent {}

class IncrementEvent extends CounterEvent {}

class DecrementEvent extends CounterEvent {}
```

Затем создадим сам класс BLoC. В него положим значение счетчика, которое мы будем менять, а также StreamController-ы, и потоки

```
import 'dart:async';
import 'package:flutter_hello_world/counter_event.dart';

class CounterBloc {
  int _counter = 0;

  final _counterStateController = StreamController<int>();
  StreamSink<int> get _inCounter => _counterStateController.sink;
  Stream<int> get counter => _counterStateController.stream;
  final _counterEventController = StreamController<CounterEvent>();
  Sink<CounterEvent> get counterEventSink => _counterEventController.sink;
```

Почти все вам должно быть уже известно, однако **Sink** мы еще на разбирали.

Sink – это дженерик интерфейс, который позволяет класть в себя поток данных. По окончании потока на нем нужно вызвать `close()`. Это все, что он умеет. В нашем случае, мы через него будем передавать event-ы в BLoC. А получать поток будем из потока **Stream counter**. Добавим в наш класс BLoC еще конструктор и метод `dispose`

```
CounterBloc() {
  _counterEventController.stream.listen(_eventToState);
}

void _eventToState(CounterEvent event) {
  if (event is IncrementEvent) {
    _counter++;
  } else if (event is DecrementEvent) {
    _counter--;
  } else {
    throw Exception('wrong Event type');
  }
  _inCounter.add(_counter);
}

void dispose() {
  _counterStateController.close();
  _counterEventController.close();
}
```

В функции eventToState мы преобразуем получаемый ивент в данные бизнес логики – инкрементируем или декрементируем.

Теперь, когда наш BLoC готов, добавим его в state виджета счетчика.

```
class _CounterWidgetState extends State<CounterWidget> {  
    final _bloc = CounterBloc();
```

Виджет с текстом текущего значения счетчика нужно обернуть в StreamBuilder

```
StreamBuilder(  
    stream: _bloc.counter, // на вход подаем Stream из BLoC  
    initialData: 0,  
    builder: (BuildContext context, AsyncSnapshot<int> snapshot) {  
        return Text('${snapshot.data}', // данные получаем через объект  
        AsyncSnapshot  
            style: TextStyle(fontSize: 20.0));  
    },  
,
```

В качестве параметра stream StreamBuilder-у мы передаем поток из нашего BLoC-а.

Данные получаем через объект AsyncSnapshot, который буквально представляет собой иммутабельное представление асинхронного вычисления.

Передавать события инкремента и декремента в BLoC

мы также будем через нажатия кнопок, с помощью counterEventSink

```
IconButton(  
    onPressed: () {  
        _bloc.counterEventSink.add(DecrementEvent());  
    },  
    icon: Icon(Icons.remove),  
    IconButton(  
    onPressed: () {  
        _bloc.counterEventSink.add(IncrementEvent());  
    },  
    icon: Icon(Icons.add),
```

И в завершение добавим dispose BLoC-а в методе dispose виджета, чтобы избежать утечек памяти.

```
@override  
void dispose() {  
    super.dispose();  
    _bloc.dispose();  
}
```

На этом все, мы имплементировали паттерн BLoC на простом примере. Как вы можете заметить, код класса CounterWidgetState стал гораздо лаконичнее. Особенно это будет заметно на примере более сложной бизнес-логики.

Рефакторим приложение погоды с использованием BLoC библиотеки

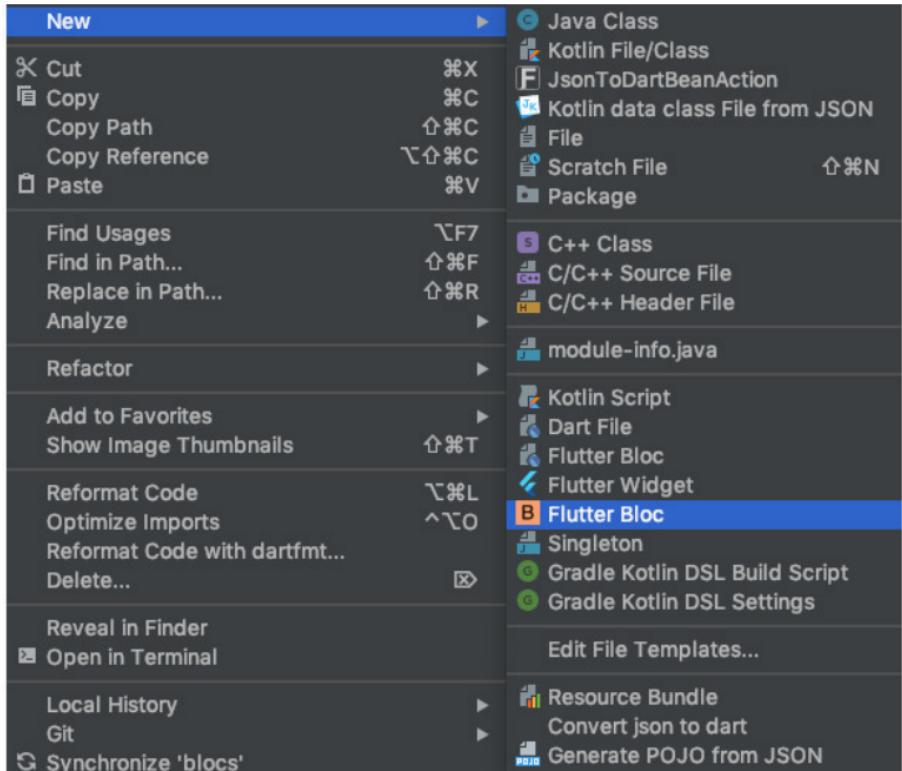
Пример со счетчиком несколько притянут за уши, поскольку бизнес логика там простейшая, и необходимость в bloc там не такая острая. А вот если мы вернемся к примеру с погодой, то паттерн bloc придется очень даже кстати.

В нашем приложении будет несколько bloc-ов; один для списка мест и второй для, собственно, погоды.

Примечание: при проектировании архитектуры приложения на Flutter надо стремиться к тому, чтобы на каждый элемент бизнес-логики был свой отдельный bloc.

Еще один можно добавить для геопозиции, однако, в нашем случае удобнее и проще использовать для местоположения InheritedWidget, поэтому код определения местоположения оставим без изменений. Итак, поехали.

Возвращаемся в ветку lesson_8_bloc из главы 8 и подключаем там bloc библиотеку аналогично тому, как описано выше. Затем создаем классы блока для погоды и мест.



Добавление нового bloc-компонентента

Для начала создадим папку **blocs**. В ней жмите правой кнопкой и выберите *New-> Flutter Bloc*. Этот элемент нам добавил плагин. Он сгенерирует базовые классы блока. Введем название блока **Weather** и **OK**. Теперь можно редактировать.

Для начала зададим классы событий. Мы будем наследовать класс **Equatable** из пакета **equatable: ^0.6.1**. Он реализи-

зовывает == и hashCode на основе props

```
part of 'weather_bloc.dart';

@Injectable()
abstract class WeatherEvent extends Equatable {
  const WeatherEvent();
}

/// запрос получения погоды
class FetchWeather extends WeatherEvent {
  final Placemark placemark;

  const FetchWeather({@required this.placemark}) : assert(placemark != null);

  @override
  List<Object> get props => [placemark];
}

/// обновление погоды
class RefreshWeather extends WeatherEvent {
  final Placemark placemark;

  const RefreshWeather({@required this.placemark}) : assert(placemark != null);

  @override
  List<Object> get props => [placemark];
}
```

Далее подготовим классы состояний

```
part of 'weather_bloc.dart';

@immutable
abstract class WeatherState extends Equatable {
  const WeatherState();

  @override
  List<Object> get props => [];

  /// пустой список
  class WeatherEmpty extends WeatherState {}

  /// в процессе загрузки
  class WeatherLoading extends WeatherState {}

  /// погода успешно загружена
  class WeatherLoaded extends WeatherState {
    final ForecastResponse forecastResponse;

    const WeatherLoaded({@required this.forecastResponse}) :
      assert(forecastResponse != null);

    @override
    List<Object> get props => [forecastResponse];
  }

  /// ошибка
  class WeatherError extends WeatherState {}
}
```

Теперь, когда у нас готовы и события, и состояния, остается только замаппить одно в другое в классе блока. Также в класс блока мы через конструктор будем передавать инстанс заготовленного в 8 главе репозитория, с помощью которого мы будем получать погоду.

```
class WeatherBloc extends Bloc<WeatherEvent, WeatherState> {
    final WeatherRepository weatherRepository;

    WeatherBloc({@required this.weatherRepository})
        : assert(weatherRepository != null);

    @override
    WeatherState get initialState => WeatherEmpty();

    @override
    Stream<WeatherState> mapEventToState(WeatherEvent event) async* {
        if (event is FetchWeather) {
            yield WeatherLoading();
            try {
                final ForecastResponse response =
                    await weatherRepository.getWeather(event.placemark);
                yield WeatherLoaded(forecastResponse: response);
            } catch (_) {
                yield WeatherError();
            }
        }
        if (event is RefreshWeather) {
            try {
                final ForecastResponse response =
                    await weatherRepository.getWeather(event.placemark);
                yield WeatherLoaded(forecastResponse: response);
            } catch (_) {
                yield WeatherError();
            }
        }
    }
}
```

Блок готов. Остается инициализировать его при старте приложения, а также имплементировать чтение и запись в него данных на экране погоды.

Добавим в класс main строку

```
BlocSupervisor.delegate = BlocDelegate();
```

Мы инициализировали делегат блока. Он будет управлять всеми блоками в приложении. Затем подготовим репозито-

рий мест.

```
final WeatherRepository weatherRepository = WeatherRepository(  
    weatherApiClient: WeatherApiClient(  
        httpClient: http.Client(),  
    ),  
)
```

И обернем приложение в **BlocProvider**

```
runApp(BlocProvider(  
    builder: (context) =>  
        WeatherBloc(weatherRepository: weatherRepository),  
    child: MyApp(),  
) );
```

С **main.dart** мы закончили. Переместимся в **weather_forecast_page.dart** и напишем в нем функцию отрисовки по состояниям:

```
Widget get _contentView {
    return BlocBuilder<WeatherBloc, WeatherState>(builder: (context,
state) {
    if (state is WeatherEmpty) {
        return errorView(context, 'No data received. Pull to refresh');
    }
    if (state is WeatherLoading) {
        return loadingView();
    }
    if (state is WeatherError) {
        return errorView(context, "Exception while fetching weather");
    }
    if (state is WeatherLoaded) {
        final weatherResponse = state.forecastResponse;
        if (weatherResponse.cod == 200.toString()) {
            initWeatherWithData(weatherResponse.list);
            return _weatherListView;
        } else {
            // В случае ошибки показываем ошибку
            Scaffold.of(context).showSnackBar(SnackBar(
                content:
                    Text("Error ${weatherResponse.cod}
${weatherResponse.message}"),
            ));
            return errorView(
                context, "Error occurred while loading data from server");
        }
    }
    return null;
});
}
```

В случае успешной загрузки данных о погоде строим ListView, обернутый в RefreshIndicator. В функции onRefresh мы будем уведомлять блок о событии обновления.

```
Widget get _weatherListView {
    return RefreshIndicator(
        onRefresh: () {
            BlocProvider.of<WeatherBloc>(context).add(
                RefreshWeather(placemark: _placemark), // добавляем в блок
                событие обновления погоды
            );
            return _refreshCompleter.future;
        },
        child: ListView.builder(
            itemCount: _weatherForecast == null ? 0 :
            _weatherForecast.length,
            itemBuilder: (BuildContext context, int index) {
                final item = _weatherForecast[index];
                if (item is WeatherListBean) {
                    return WeatherListItem(item);
                } else if (item is DayHeading) {
                    return HeadingListItem(item);
                } else
                    throw Exception("wrong type");
            }));
}
```

Также добавим функцию **refreshWrapper** в виде обертки над **BlocListener**, которая позволит нам обновлять погоду с помощью **Completer**-а.

```
Widget get _refreshWrapper {
    return BlocListener<WeatherBloc, WeatherState>(
        listener: (context, state) {
            if (state is WeatherLoaded) {
                _refreshCompleter?.complete();
                _refreshCompleter = Completer();
            }
        },
        child: _contentView);
}
```

Мы подготовили UI, остается загрузить, собственно, данные. Для этого изменим функцию **getWeather** следующим образом:

```
void _getWeather() {
    _initPlaceMark();
    if (_placemark != null) {
        BlocProvider.of<WeatherBloc>(context)
            .add(FetchWeather(placemark: _placemark)); // добавляем в
        блок событие загрузки погоды
    }
}
```

Загрузка погоды на этом готова, остается добавить блок для списка мест. Аналогично создаем классы для событий и для состояний.

Сначала классы событий:

```
part of 'places_bloc.dart';

@immutable
abstract class PlacesEvent extends Equatable {
    const PlacesEvent();
}

class FetchPlaces extends PlacesEvent {
    const FetchPlaces();

    @override
    List<Object> get props => [];
}

class AddPlaceEvent extends PlacesEvent {
    final Placemark placemark;

    const AddPlaceEvent({@required this.placemark}) : assert(placemark != null);

    @override
    List<Object> get props => [placemark];
}

class RemovePlaceEvent extends PlacesEvent {
    final PlacemarkLocal placemarkLocal;

    const RemovePlaceEvent({@required this.placemarkLocal}) :
        assert(placemarkLocal != null);

    @override
    List<Object> get props => [placemarkLocal];
}
```

И затем классы состояний:

```
part of 'places_bloc.dart';

@immutable
abstract class PlacesState extends Equatable {
  const PlacesState();

  @override
  List<Object> get props => [];

}

class EmptyPlacesState extends PlacesState {}

class LoadingPlacesState extends PlacesState {}

class LoadedPlacesState extends PlacesState {
  final List<PlacemarkLocal> placemarks;

  const LoadedPlacesState({@required this.placemarks})
    : assert(placemarks != null);

  @override
  List<Object> get props => [placemarks];
}

class ErrorPlacesState extends PlacesState {}

class ErrorAddingPlaceState extends PlacesState {}

class RemovedPlaceState extends PlacesState {
  final int id;

  const RemovedPlaceState({@required this.id}) : assert(id > 0);

  @override
  List<Object> get props => [id];
}

class ErrorRemovingPlaceState extends PlacesState {}
```

В PlacesBloc передаем репозиторий мест, подготовленный в главе 8. В нем мы в соответствующих методах преобразуем получаемые события в состояния:

```

part 'places_event.dart';
part 'places_state.dart';

class PlacesBloc extends Bloc<PlacesEvent, PlacesState> {
    final PlacesRepository placesRepository;

    PlacesBloc({@required this.placesRepository})
        : assert(placesRepository != null);

    @override
    PlacesState get initialState => EmptyPlacesState();

    @override
    Stream<PlacesState> mapEventToState(PlacesEvent event) async* {
        if (event is FetchPlaces) {
            yield LoadingPlacesState();
            try {
                final List<PlacemarkLocal> places =
                    await placesRepository.getPlaces();
                yield LoadedPlacesState(placemarks: places);
            } catch (_) {
                yield ErrorPlacesState();
            }
        }

        if (event is AddPlaceEvent) {
            yield LoadingPlacesState();
            try {
                /// add new place
                await placesRepository.addPlace(event.placemark);
                /// get all new places as list to reinit UI
                final List<PlacemarkLocal> places =
                    await placesRepository.getPlaces();
                yield LoadedPlacesState(placemarks: places);
            } catch (_) {
                yield ErrorAddingPlaceState();
            }
        }
        if (event is RemovePlaceEvent) {
            try {
                /// remove new place
                await placesRepository.deletePlacemark(event.placemarkLocal.id);
                yield RemovedPlaceState(id: event.placemarkLocal.id);
            } catch (_) {
                yield ErrorRemovingPlaceState();
            }
        }
    }
}

```

Теперь нам нужно поменять инициализацию BlocProvider в main.dart. Для начала добавим строку

```
final PlacesRepository placesRepository = PlacesRepository();
```

Затем мы заменим BlocProvider на MultiBlocProvider, ведь у нас теперь два блока, а не один.

```
runApp(MultiBlocProvider(
  providers: [
    BlocProvider<WeatherBloc>(
      builder: (context) => WeatherBloc(weatherRepository),
    ),
    BlocProvider<PlacesBloc>(
      builder: (context) => PlacesBloc(placesRepository),
    ),
  ],
  child: MyApp(),
));
```

Переходим в places_page.dart и переделываем код получения и редактирования списка мест. В методе инициализации состояния добавляем событие подгрузки списка мест.

```
@override
void initState() {
  super.initState();
  BlocProvider.of<PlacesBloc>(context).add(FetchPlaces());
}
```

А функция создания контента будет выглядеть следующим образом:

```
Widget get _contentView {
    return BlocBuilder<PlacesBloc, PlacesState>(builder: (context,
    state) {
        if (state is EmptyPlacesState) {
            return errorView(
                context, 'No places yet. Tap Add button to create one');
        }
        if (state is LoadingPlacesState) {
            return loadingView();
        }
        if (state is ErrorPlacesState) {
            return errorView(context, "Exception while reading places");
        }
        if (state is ErrorAddingPlaceState) {
            showSnackBar(context, _scaffoldKey, "Exception while adding
place");
        }
        if (state is LoadedPlacesState) {
            _placemarkList = state.placemarks;
        }
        if (state is RemovedPlaceState) {
            var index = -1;
            for (int i = 0; i < _placemarkList.length; i++) {
                if (_placemarkList[i].id == state.id) {
                    index = i;
                    break;
                }
            }
            if (index > 0) {
                _placemarkList.removeAt(index);
            }
        }
        if (state is ErrorRemovingPlaceState) {
            showSnackBar(context, _scaffoldKey, "Exception while deleting
place");
        }
    });
}
```

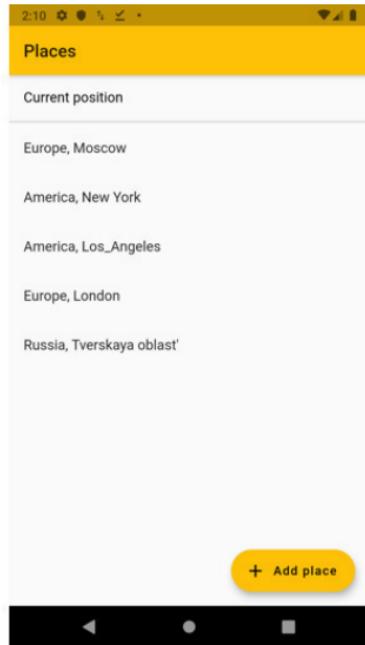
Функции добавления и удаления также будут работать с блоком:

```
/// Обработчик нажатия на плавающую кнопку – добавление нового места
void _onAddNew() async {
    final result = await Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => MapPage()),
    ); // ждем добавленное место

    BlocProvider.of<PlacesBloc>(context).add(AddPlaceEvent(placemark: result));
}

/// обработчик удаления элемента из списка
void _onRemoveItem(int index) {
    BlocProvider.of<PlacesBloc>(context)
        .add(RemovePlaceEvent(placemarkLocal: _placemarksList[index]));
}
```

На этом все, запускаем!



Список мест

С точки зрения пользователя, ничего не поменялось, однако мы значительно отрефакторили код, и этим облегчили себе жизнь – наш код стал более поддерживаемым и обрел архитектуру.

Урок 10. DI, Тесты

В этой главе:

- *Dependency Injection во Flutter*
- *Внедряем зависимости в примере с погодой*
- *Unit тесты*
- *UI тесты*
- *Интеграционное тестирование*

Dependency Injection во Flutter

Мы уже знаем как писать приложения на Flutter. В этом заключительном уроке мы изучим еще один важный, неосвещенный ранее, аспект разработки ПО – тестирование. Как программисты, вы должны знать, что тесты подразделяются на три типа: *unit-тесты*, *UI-тесты*, и *интеграционные тесты*. Дальше в этой главе мы рассмотрим их поочереди в контексте Flutter-а. Однако, прежде нам нужно подготовить наш код. Для того, чтобы протестировать отдельный участок приложения, необходимо чтобы объекты, от которых этот участок зависит, инстанцировались вовне. Так мы сможем подставлять замоканные (тестовые болванки) зависимости тестируемому объекту. При этом мы приведем наш код в соответствие с принципом единственной обязанности – *Single Responsibility*. В соответствии с ним, компонент –

функция, класс или модуль – должен отвечать за что-то одно, а все зависимости этого компонента должны создаваться вовне и предоставляться (provides) в этот компонент извне.

Хватит теории на этом, попробуем на практике.

Для Flutter и Dart написано уже несколько библиотек DI, вы можете найти и использовать любую из них на сайте pub.dev. В этом курсе мы подключим простой и легковесный пакет **injector** (<https://pub.dev/packages/injector>):

```
dependencies:  
  injector: ^1.0.8
```

Выполним команду *flutter pub get* после изменения **pubspec.yaml** и идем в **main.dart**

В методе **main()** добавьте вызов функции **initInjector()**; перед созданием **SimpleBlocDelegate** и затем объявим сам метод:

```
void initInjector() {
    // получаем статический инстанс инжектора
    Injector injector = Injector.appInstance;
    // регистрируем объект http.Client-а в дереве зависимостей
    injector.registerSingleton<http.Client>((injector) {
        return http.Client();
    });
    // регистрируем объект WeatherApiClient-а в дереве зависимостей
    injector.registerSingleton<WeatherApiClient>((injector) {
        var httpClient = injector.getDependency<http.Client>();
        return WeatherApiClient(httpClient: httpClient);
    });
    // регистрируем объект WeatherRepository-а в дереве зависимостей
    injector.registerSingleton<WeatherRepository>((injector) {
        var webApiClient = injector.getDependency<WeatherApiClient>();
        return WeatherRepository(weatherApiClient: webApiClient);
    });
    // регистрируем объект PlacesRepository-а в дереве зависимостей
    injector.registerSingleton<PlacesRepository>((injector) {
        return PlacesRepository();
    });
}
```

Теперь поменяем немного код BloC-ов, внедряя зависимость с помощью конструкторов:

```
class WeatherBloc extends Bloc<WeatherEvent, WeatherState> {
    WeatherRepository weatherRepository;

    WeatherBloc() {
        Injector injector = Injector.appInstance;
        weatherRepository = injector.getDependency<WeatherRepository>();
    }
    *
    *

    class PlacesBloc extends Bloc<PlacesEvent, PlacesState> {
        PlacesRepository placesRepository;

        PlacesBloc() {
            Injector injector = Injector.appInstance;
            placesRepository = injector.getDependency<PlacesRepository>();
        }
        *
        *
```

Готово. Теперь наш код построен по принципу инверсии зависимостей! А мы можем полностью переключится на тесты.

Unit тесты

Unit тесты позволяют протестировать небольшой атомарный участок кода: функцию, класс, метод. Для того, чтобы имплементировать unit тесты в приложении, нужно заимпортировать package **test**. В нем содержаться базовая функциональность для тестирования кода на языке Dart, без учета специфики Flutter. Вернемся к примеру со счетчиком.

По умолчанию в *pubspec.yaml* уже есть импорт тестов:

```
dev_dependencies:  
  flutter_test:  
    sdk: flutter
```

Поэтому, оставим *pubspec.yaml* без изменений, а в папке *test* создадим файл *counter_test.dart* с содержимым:

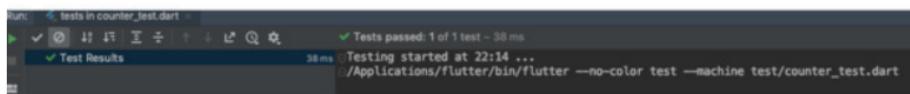
```
import 'package:flutter_hello_world/counter_bloc.dart';  
import 'package:flutter_test/flutter_test.dart';  
  
void main() {  
  CounterBloc counterBloc;  
  
  /// вызывается до начала тестов  
  setUp(() {  
    counterBloc = CounterBloc();  
  });  
  
  /// вызывается после окончания тестов  
  tearDown(() {  
    counterBloc.dispose();  
  });  
}
```

Мы заимпортили пакет test, а также подготовили функции setup и teardown, которые вызываются соответственно до и после тестов. Также мы добавили объект блока счетчика, над которым и будем производить тестирование.

Добавим, собственно, саму функцию теста:

```
test('Counter test', () {
  counterBloc.counterEventSink.add(IncrementEvent());
  expect(counterBloc.counter, emits(1));
});
```

Здесь мы отправляем событие инкремента, и ждем, что bloc выдаст единичку. Вот и все. Жмем правой кнопкой по файлу с тестом и в выпадающем меню находим Run Tests in *counter_test.dart*. Нажимаем и смотрим на результаты.



Результат запуска unit теста

Тест пройден.

Mockito

Снова вернемся в ветку с приложением погоды и создадим тест для класса репозитория погоды – WeatherRepository. Для того чтобы сымитировать (замокать) сервер, нам понадобится package **mockito**:

```
dev_dependencies:  
  flutter_test:  
    sdk: Flutter  
  mockito: 4.1.1
```

При наследовании объекта от класса Mock все объявленные и неимплементированные методы получат имплементацию. То есть, если в классе объявлена функция Future <Response> getWeather, то в замоканном виде объект будет возвращать Future с ответом. Мы замокаем http клиент. Для этого нужно наследовать или примешивать класс Mock, а также имплементировать целевой класс:

```
import 'package:flutter_test/flutter_test.dart';  
import 'package:mockito/mockito.dart';  
import 'package:http/http.dart' as http;  
  
class MockHttpClient extends Mock implements http.Client {}  
  
main() {  
  setUp(() {});  
  tearDown(() {});  
}
```

Добавим объекты репозитория и необходимого для него

WeatherApiClient.

```
class MockAPIClient extends Mock implements WeatherApiClient {}  
  
main() {  
    WeatherRepository weatherRepository;  
  
    setUp(() {  
        final httpClient = MockHttpClient();  
        final apiClient = MockAPIClient();  
        weatherRepository = WeatherRepository(weatherApiClient:  
            apiClient);  
    });
```

Для имитации ответа мы просто скопируем пример ответа сервера с сайта openWeather: <https://openweathermap.org/forecast5> в константу:

```
const String successJson =  
'{"city":{"id":1851632,"name":"Shuzenji","coord":{"lon":138.933334,"la  
t":34.966671}, "country": "JP", "timezone": 32400,  
"cod":200,"message":0.0045,"cnt":38,"list":[{"dt":1406106000,"main":  
{"temp":298.77,"temp_min":298.77,"temp_max":298.774,"pressure":1005.93  
,"sea_level":1018.18,"grnd_level":1005.93,"humidity":87,"temp_kf":0.26  
}],"weather":[{"id":804,"main":"Clouds","description":"overcast  
clouds","icon":"04d"}],"clouds":{"all":88},"wind":{"speed":5.71,"deg":  
229.501}, "sys":{"pod":"d"}, "dt_txt":"2014-07-23 09:00:00"}]';
```

Также добавим тестовый объект места, и инициализируем переменную Uri:

```
var testPlacemark = Placemark(  
    name: 'test',  
    country: 'test',  
    position: Position(longitude: 0, latitude: 0));  
  
var uri;
```

Метод setup будет выглядеть следующим образом:

```
setUp() {
    mockClient = MockHttpClient();
    apiClient = WeatherApiClient(httpClient: mockClient);
    weatherRepository = WeatherRepository(weatherApiClient: apiClient);

    double lat = testPlacemark?.position?.latitude;
    double lng = testPlacemark?.position?.longitude;

    var queryParameters = {
        // подготавливаем параметры запроса
        'APPID': Constants.WEATHER_APP_ID,
        'units': 'metric',
        'lat': lat.toString(),
        'lon': lng.toString(),
    };

    uri = Uri.https(Constants.WEATHER_BASE_URL_DOMAIN,
        Constants.WEATHER_FORECAST_PATH, queryParameters);
}:
```

Теперь нам нужно сымитировать успешный и неуспешный ответ сервера. Для того, чтобы запускать сразу несколько тестов, можно использовать функцию **group**
Добавим для начала в нее тест успешного запроса

```
test('success response test', () async {
    // Возвращаем успешный результат
    when(mockClient.get(uri)).thenAnswer(
        (_) async => http.Response(successJson, 200, headers: {
            HttpHeaders.contentTypeHeader: 'application/json;
            charset=utf-8'
        });
    );

    var response = await weatherRepository.getWeather(testPlacemark);
    // проверяем, что ответ пришел в виде [ResponseWrapper]
    expect(response, isInstanceOf<ResponseWrapper>());
    // и содержит корректный [ForecastResponse]
    expect(response.forecastResponse, isInstanceOf<ForecastResponse>());
    expect(response.errorResponse, null);
    expect(response.forecastResponse.city.name, "Shuzenji");
});|
```

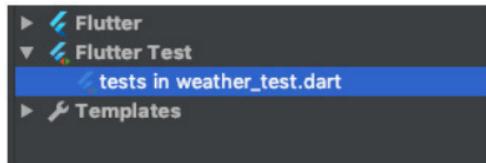
Как мы уже знаем, `when` задает правила. В нашем случае правило такое: «*При попытке запроса по совпадающему `uri` выдавать подготовленный json `successJson`*». Далее, в блок-

как эксперт мы проверяем наш ответ и его содержимое на соответствие требуемому типу класса и корректность данных.

Добавим аналогичный тест на случай ошибочного запроса:

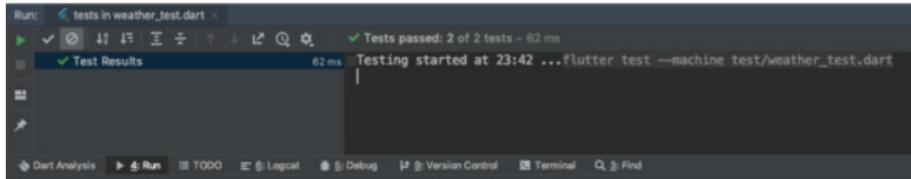
```
test('returns error message', () async {
    // Возвращаем неуспешный результат
    when(mockClient.get(uri)).thenAnswer(
        (_) async => http.Response('{"message": "Not Found"}', 404,
    headers: {
        HttpHeaders.contentTypeHeader: 'application/json;
        charset=utf-8';
    }));
    var response = await weatherRepository.getWeather(testPlacemark);
    // проверяем, что ответ пришел в виде [ResponseWrapper]
    expect(response, isInstanceOf<ResponseWrapper>());
    // и содержит корректный [ErrorResponse]
    expect(response.errorResponse, isInstanceOf<ErrorResponse>());
    expect(response.forecastResponse, null);
    expect(response.errorResponse.message, "Not Found");
});
```

Теперь в меню Run Configurations выберите Flutter Test и запустите



Запуск unit тестов с помощью IDE

Вы должны увидеть сообщение Tests passed



Результат запуска Unit теста

UI тесты

UI тесты во Flutter именуются Widget-тестами, и суть их заключается в том, чтобы убедиться, что отдельный виджет выглядит и ведет себя так, как от него ожидается. Виджеты имеют свой жизненный цикл, реагируют на действия пользователя, отрисовываются сами, а также могут содержать внутри себя другие виджеты. Поэтому, UI-тесты более сложные, чем Unit. Приступим к их изучению.

В `pubspec.yaml` у нас уже имеется нужный импорт пакета – `flutter_test`

В нем содержатся нужные нам компоненты:

WidgetTester – класс, который позволяет создавать виджеты и взаимодействовать с ними в тестовой среде.

testWidgets – функция, в которой мы будем писать сам тест. Эта функция будет использоваться вместо функции `test`, которую мы писали для unit-тестов.

Finder и константы **Matcher** – позволяет найти виджет определенного класса и отделить его среди других виджетов этого же класса.

Вернемся к примеру со счетчиком в ветку `lesson_3_counter` и создадим в папке `test` файл `counter_widget_test.dart`:

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

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

void main() {
    // функция testWidgets будет запускать тесты и предоставляет объект
    // WidgetTester,
    // который позволяет создавать виджеты
    testWidgets('Counter widget test', (WidgetTester tester) async {
        // тесты напишем здесь
    });
}
```

Создадим виджет нашего приложения со счетчиком.
Для этого используем функцию **pumpWidget** класса **WidgetTester**

```
await tester.pumpWidget(MyApp());
```

Для управления жизненным циклом виджета, например, в случае, если нам нужно будет перерисовать виджет, или чтобы вызвать на нем **build** снова в тестовой среде, используйте методы **pump ()** и **pumpAndSettle ()**

Отличие их в том, что **pump ()** – вызывает виджета **build** один раз, в то время, как **pumpAndSettle ()** вызывает отрисовку до тех пор, пока в очереди не закончатся фреймы анимации.

Чтобы после создания виджета получить доступ к его свойствам и к нему самому, нужно использовать функции **find**. Итак, найдем виджет с текстом – значением счетчика, и убедимся, что оно 0 на старте

```
expect(find.text("0"), findsOneWidget); // находит виджет  
expect(find.text("1"), findsNothing); // не находит
```

Далее убедимся, что у нас есть виджет счетчика и тапнем по нему:

```
expect(find.byIcon(Icons.add), findsOneWidget);  
await tester.tap(find.byIcon(Icons.add));
```

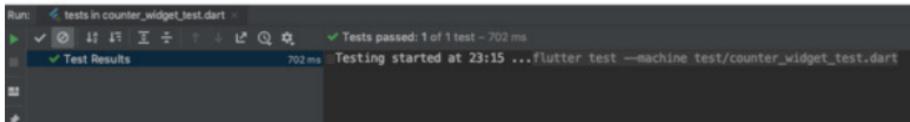
Теперь, поскольку состояние (state) виджета изменилось, нужно вызвать *pump()*, чтобы у виджета вызвался метод *build*

```
await tester.pump();
```

И остается убедиться, что значение увеличилось:

expect (find. text («1»), findsOneWidget);

Запускаем!



Результат запуска UI теста

Весь код теста:

```
import 'package:flutter/material.dart';
import 'package:flutter_hello_world/main.dart';
import 'package:flutter_test/flutter_test.dart';

void main() {
    // функция testWidgets будет запускать тесты и предоставляет объект
    WidgetTester,
    // который позволяет создавать виджеты
    testWidgets('CounterWidget test', (WidgetTester tester) async {
        await tester.pumpWidget(MyApp());

        // Убедимся, что значение счетчика на старте равно 0, а не единице
        expect(find.text("0"), findsOneWidget);
        expect(find.text("1"), findsNothing);

        // Убедимся, что присутствует кнопка "плюса"
        expect(find.byIcon(Icons.add), findsOneWidget);

        // жмем на кнопку добавить
        await tester.tap(find.byIcon(Icons.add));
        // ждем отрисовки виджета после нажатия кнопки
        await tester.pump();

        // Убедимся, что значение счетчика увеличилось до единицы
        expect(find.text("1"), findsOneWidget);

    });
}
```

Интеграционные тесты

Интеграционные тесты – это буквально объединенные тесты, то есть сложные, выполняемые над всем приложением или подсистемой сразу. Такой тест как правило подразумевает запуск приложения на реальном устройстве или эмуляторе, выполнение действий бизнес-логики, такие как запрос на сервер и/или запись в БД. Интеграционные тесты могут быть особенно полезны для автоматического тестирования в CI (Continuous Integration) – они могут запускаться на build-сервере для каждой сборки автоматически или по расписанию, проверяя заданные разделы приложения, не требуя участия людей-тестировщиков.

Итак, приступим. Интеграционные тесты запускаются как отдельный процесс, поэтому последовательность действий написания тестов тоже будет немного отличаться. Добавим в pubspec.yaml строки

```
dev_dependencies:  
    flutter_driver:  
        sdk: flutter  
    test: any
```

Затем создадим папку **test-driver**, а в ней файлы **app.dart** и **app-test.dart**

Первый файл – **app. dart** – будет представлять инструментальную (запускаемую с помощью девайса или эмулятора) версию приложения. Второй файл – **app-test. dart** – будет содержать, собственно, сам тест. В app. dart напишите следующее:

```
import 'package:flutter_driver/flutter_driver_extension.dart';
import 'package:flutter_hello_world/main.dart' as app;

void main() {
    // Подключаем driver_extension
    enableFlutterDriverExtension();

    // Вызываем функцию `main()` нашего приложения
    app.main();
}
```

Прежде, чем писать тест, добавим в файл *places_page. dart* для первого элемента списка, который показывает погоду для текущего местоположения, ключ:

```
child: Text("Current position",
key: Key('current_position'),
```

Теперь в файл *app_test. dart* добавим код

```
import 'package:flutter_driver/flutter_driver.dart';
import 'package:test/test.dart';

void main() {
  group('Weather app test', () {
    // С помощью Finder-ов и находим виджет Text с ключом
    'current_position'
    final currentPositionTextFinder =
      find.byValueKey('current_position');

    FlutterDriver driver;

    // Подключаемся к Flutter driver-y
    setUpAll(() async {
      driver = await FlutterDriver.connect();
    });

    // После выполнения теста отключаемся
    tearDownAll(() async {
      if (driver != null) {
        driver.close();
      }
    });
  });

  test('Текущее местоположение', () async {
    // Ждем 3 секунды пока загрузится список мест
    sleep(Duration(seconds: 3));

    // Убеждаемся, что виджет содержит соответствующий текст
    expect(await driver.getText(currentPositionTextFinder), "Current
position");
  });
}
```

И запустим из терминала с помощью команды

```
flutter drive --target=test_driver/app.dart
```

Примечание: У вас должно быть подключено устройство или запущен эмулятор.

Тест драйвер сам установит и стартует приложение на эмуляторе: и убедится, что тесты отрабатывают. Смотрим

на эмулятор и убеждаемся, что тест выполняет свою работу, а в логах наблюдаем успешность прохождения теста:

```
Using device Android SDK built for x86.
Starting application: test_driver/app.dart
Initializing gradle...
Resolving dependencies...                                2.1s
Installing build/app/outputs/apk/app-debug.apk...          3.6s
Running Gradle task 'assembleDebug'...
Running Gradle task 'assembleDebug'... Done                6.8s
Built build/app/outputs/apk/debug/app-debug.apk.
[flutter] Observatory listening on http://127.0.0.1:35125/cartheXWnAv/
00:00 +0: Weather app test (setUpAll)
[info ] FlutterDriver: Connecting to Flutter application at http://127.0.0.1:35704/cartheXWnAv/
[trace] FlutterDriver: Isolate found with number: 81222493966859
[trace] FlutterDriver: Isolate is paused at start.
[trace] FlutterDriver: Attempting to resume isolate
[trace] FlutterDriver: Waiting for service extension
[flutter] [0841]: Transition { currentState: EmptyPlacesState, event: FetchPlaces, nextState: LoadingPlacesState }
[info ] FlutterDriver: Connected to Flutter application.
00:01 +0: Weather app test TestQuery местоположение
[flutter] [0841]: Transition { currentState: LoadingPlacesState, event: FetchPlaces, nextState: LoadedPlacesState }
00:05 +1: Weather app test Заряжая новую зону recursive места
[flutter] [0841]: Transition { currentState: WeatherEmpty, event: FetchWeather, nextState: WeatherLoading }
[flutter] [0841]: Transition { currentState: WeatherLoading, event: FetchWeather, nextState: Weatherloaded }
[flutter] [0841]: Location with the name "Russia" doesn't exist
00:10 +2: Weather app test (tearDownAll)
00:10 +2: All tests passed!
Stopping application instance.
```

Результат запуска интеграционного теста

Тест прошел успешно. Но это был простой тест. Напишем более сложный и приближенный к реальному миру. Он будет проверять, что по нажатию на кнопку Current Position будет и открываться экран «погода», и она будет загружаться с сервера. Для этого добавим второй тест в группу. В нем мы будем нажимать на кнопку current position, затем ждать подгрузки погоды, убеждаться, что у нас подгрузился список в ListView и сам список скроллится вниз до n-ого элемента.

```

test('Загрузка погоды для текущего места', () async {
  // Нажимаем по виджету текущей погоды
  await driver.tap(currentPositionTextFinder);
  // ждем 5 секунд пока погода и экран загрузятся
  sleep(Duration(seconds: 5));

  // Находим список с погодой
  final listView = find.byValueKey('weather_listview');
  // Поскольку у нас прогноз состоит из 5 дней по 7 элементов в
  // каждом,
  // то в списке будет как минимум 30 строк, и 30-ая строка будет в
  // конце
  final thirtyElement = find.byValueKey('weatherListItem_30');

  // убеждаемся, что список прокручивается до третьего дня с погодой
  await driver.scrollUntilVisible(
    // Указываем список в качестве параметра
    listView,
    // И элемент, который мы ищем
    thirtyElement,
    // С помощью отрицательного значения dyScroll скроллим список вниз
    dyScroll: -200.0,
  );
  sleep(Duration(seconds: 2)); // просто ждем, чтобы осознать свой
  успех
});

```

Запускаем той же командой и смотрим в terminal.

```

Using device Android SDK built for x86.
Starting application: test_driver/app.dart
Initializing gradle...                                2.1s
Resolving dependencies...                            16.2s
Installing build/app/outputs/apk/app.apk...          3.6s
Running Gradle task 'assembleDebug'...
Running Gradle task 'assembleDebug'... Done           6.8s
Built build/app/outputs/apk/debug/app-debug.apk.
I/flutter (18641): Observatory listening on http://127.0.0.1:35125/JcarheXWjnA=/
00:00 +0: Weather app test (setUpAll)
[info] FlutterDriver: Connecting to Flutter application at http://127.0.0.1:57864/JcarheXWjnA=/
[trace] FlutterDriver: Isolate found with number: 81222493966859
[trace] FlutterDriver: Isolate is paused at start.
[trace] FlutterDriver: Attempting to resume isolate
[trace] FlutterDriver: Waiting for service extension
I/flutter (18641): Transition { currentState: EmptyPlacesState, event: FetchPlaces, nextState: LoadingPlacesState }
[info] FlutterDriver: Connected to Flutter application.
00:01 +0: Weather app test Текущее местоположение
I/flutter (18641): Transition { currentState: LoadingPlacesState, event: FetchPlaces, nextState: LoadedPlacesState }
00:05 +1: Weather app test Загрузка погоды для текущего места
I/flutter (18641): Transition { currentState: WeatherEmpty, event: FetchWeather, nextState: WeatherLoading }
I/flutter (18641): Transition { currentState: WeatherLoading, event: FetchWeather, nextState: WeatherLoaded }
I/flutter (18641): Location with the name "Russia/" doesn't exist
00:18 +2: Weather app test (tearDownAll)
00:18 +2: All tests passed!
Stopping application instance.

```

Результат запуска интеграционного теста – Успех!

Заключение

Вы освоили базовые навыки, необходимые для разработки мобильных приложений на Flutter. Примите мои поздравления!

FlutterTM – относительно молодая технология, но с каждым годом она становится все сильнее, а ряды Flutter разработчиков пополняются все новыми и новыми программистами. Язык Dart понятный, несложный, а фреймворк FlutterTM удобен в использовании, и позволяет писать красивые и быстродействующие приложения сразу на две платформы.

Мы не знаем наверняка каким будет мир мобильной разработки через 5—10 лет. Станет ли Flutter основным инструментом для создания кроссплатформенных приложений? Вполне возможно. Однако уже сегодня с помощью него можно писать по-настоящему кроссплатформенный код для iOS и Android в продакшне.

В то же время, ничто не стоит на месте, и код для мобильных приложений на Java и Objective-C уже почти никто не пишет. Многие переходят или уже полностью перешли на Kotlin и Swift, которые более легковесны, и на которых легче, удобнее и быстрее писать. Аналогично, никто сейчас не станет писать фронтенд на Fortran или чистом C.

Даже на чистом JS+CSS+HTML уже мало кто пишет. Vue, JS, TypeScript и многие другие фреймворки и новые языки облегчили жизнь фронт-енд разработчикам. Мир меняется, меняются языки и платформы. Весьма вероятно, что через 5 лет все будут писать на Dart, или другом едином языке, под новые устройства с универсальными интерфейсами. В любом случае, нас ждет мир увлекательных новшеств и решений, а я желаю Вам не пропустить появление перспективных технологий и инструментов, но успеть изучить их и применить на практике для крутых успешных проектов. Удачи!

Если у вас остались какие-либо вопросы, или пожелания, смело задавайте их на форуме проекта **FlyFlutter.ru** в разделе, посвященном этой книге.

Полезные ссылки

Весь код к этой книге находится в репозитории на github:
https://github.com/acinonyxjubatus/flyflutter_fast_start

Форум проекта: **forum.flyflutter.ru**

В книге используются фрагменты исходного кода фреймворка Flutter™. Лицензия доступна по адресу <https://github.com/flutter/flutter/blob/master/LICENSE>

Flutter and the related logo are trademarks of Google LLC.

We are not endorsed by or affiliated with Google LLC.

Урок 1. Запускаем Flutter

Код для главы в ветке *lesson_1_hello_world*

Урок 2. Язык программирования Dart

Для более подробного знакомства с языком, всеми его правилами ключевыми словами, операндами, рекомендуется посетить официальный сайт языка: <https://dart.dev/>

Подробная статья про компиляцию на Flutter: <https://proandroiddev.com/flutters-compilation-patterns-24e139d14177>

Урок 3. StatelessWidget и StatefulWidget

Код для главы в ветках *lesson_3_1_stateless*,

lesson_3_1_stateful

<https://flutter.dev/docs/development/ui/layout>

Туториал по созданию: Stateless виджета <https://medium.com/flutter/how-to-create-stateless-widgets-6f33931d859>

Урок 4. Создание списка элементов

Код для главы в ветке: ***lesson_4_listview***

Использование пакетов и библиотек во Flutter приложениях: <https://flutter.dev/docs/development/packages-and-plugins/using-packages>

Статьи по ListView: <https://medium.com/@DakshHub/flutter-displaying-dynamic-contents-using-listview-builder-f2cedb1a19fb>

<https://fidev.io/flutter-listview/>

<https://flutter.dev/docs/cookbook/lists/mixed-list>

<https://api.flutter.dev/flutter/widgets/ListView-class.html>

Урок 5. Загрузка данных с сервера

Код для главы в ветке: ***lesson_5_http***

Маппинг данных: <https://flutter.dev/docs/cookbook/networking/background-parsing>

Asynch / Await: <https://www.youtube.com/watch?v=SmTCmDMi4BY>

Streams: <https://www.youtube.com/watch?v=nQBpOIHE4eE>

Загрузка данных: <https://flutter.dev/docs/cookbook/networking/fetch-data>

Asynch / Await: <https://dart.dev/codelabs/async-await>

Урок 6. Inherited Widgets, Elements, Keys

Ветка этого урока: [*lesson_6_inherited*](#)

Inherited Widgets: <https://ericwindmill.com/using-flutter-inherited-widgets-effectively>

<https://flutterbyexample.com/set-up-inherited-widget-app-state/>

https://medium.com/@mehmetf_71205/inheriting-widgets-b7ac56dbbeb1

<https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d>

<https://www.youtube.com/watch?v=Eca-RIpQrvE>

Ключи, Keys: <https://medium.com/flutter-community/elements-keys-and-flutters-performance-3ef15c90f607>

Урок 7. Навигация между экранами, Работа с Google Maps

Ветка этого урока: [*lesson_7_navigation_maps*](#)

Подключение Google Maps во Flutter: <https://codelabs.developers.google.com/codelabs/google-maps-in-flutter/#5>

<https://medium.com/flutter/google-maps-and-flutter-cfb330f9a245>

Подключение пакета timezone: <https://medium.com/flutter-community/working-with-timezones-in-flutter-1c304089dcf9>

Урок 8. SQLite, Clean Architecture

Ветка с кодом по этой главе:

lesson_8_sqlite_clean_architecture

Подключение SQLite во Flutter: <https://flutter.dev/docs/cookbook/persistence/sqlite>

Clean architecture: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

Урок 9. BLoC, Streams

Ветка с кодом по этой главе – ***lesson_9_bloc***,
lesson_9_1_counter_bloc

Bloc library: <https://bloclibrary.dev/#/gettingstarted>

Bloc library статья: <http://flutterdevs.com/blog/bloc-pattern-in-flutter-part-1/>

Урок 10. DI, Тесты

Ветка с кодом по этой главе – ***lesson_10_di_tests***,
lesson_9_1_counter_bloc

Инъекция зависимостей: https://en.wikipedia.org/wiki/Dependency_injection

<https://blog.usejournal.com/compile-time-dependency-injection-in-flutter-95bb190b4a71>

Тестирование: <https://flutter.dev/docs/testing>

Интеграционное тестирование: <https://flutter.dev/docs/cookbook/testing/integration/introduction>

Unit-тесты bloc: <https://medium.com/flutter-community/unit-testing-with-bloc-b94de9655d86>

Unit-тесты и интеграционные тесты, обзорная статья: <https://blog.usejournal.com/integration-and-unit-testing-in-flutter-f08e4bd961d5>