1. Кроссплатформенная разработка мобильных приложений. Плюсы и минусы данного подхода.
2. Dart. Преимущества и недостатки языка. Платформы на которых может работать Dart.
3. Dart types. Основные типы. Numbers. Strings. Booleans. Lists. Sets. Maps. Runes and grapheme clusters. Symbols.
4. Dart Class. Структура класса, типы наследования. Dart interface
5. Dart Mixins. Сравнение интерфейсов и примесей.
6. Dart functions. Параметризации функций. Типы функций.
7. Dart Future. Назначение и применение.
8. Dart Stream. Назначение и применение.
9. Dart Isolate. Создание и взаимодействие с изолятом.
10. Flutter. Свойства фреймворка. Архитектура
11. Flutter. Структура проекта.
12. Flutter Widget. Flutter State. Типы виджетов.
13. Flutter. Stateless Widget. StatefulWidget. Отличия. Применение.
14. Flutter Widget Tree. Element Tree. Render Tree.
15. Flutter. Layout Widgets. Single-child layout widgets. Align. AspectRatio. Baseline. Center. ConstrainedBox. Container. MediaQuery.
16. Flutter. Layout Widgets. Виджеты с одним потомком. Expanded. Padding. SizedBox. Transform. OverflowBox. FittedBox
17. Flutter. Layout Widgets. Виджеты с множеством потомков. Column. GridView. Stack. IndexedStack.
18. Flutter. Layout Widgets. Виджеты с множеством потомков. Row. ListView. ListTile. Table. Wrap.
19. Flutter. Компоненты пользовательского интерфейса. Appbar. RaisedButton. Scaffold. FloatingActionButton.
20. Flutter. Компоненты пользовательского интерфейса. Drawer. PopupButton. Date&Time Picker.
21. Flutter. Компоненты пользовательского интерфейса. AlertDialog. BottomSheet. ExpansionPanel. SimpleDialog. SnackBar.
22. Flutter. Компоненты пользовательского интерфейса. Card. Divider. Stepper. Chip.
23. Flutter Touch interactions. Dismissible. DragTarget. Draggable. GestureDetector. LongPressDraggable
24. Flutter. Виджеты в стиле Купертино. CupertinoActionSheet. CupertinoSegmentedControl. CupertinoPicker
25. Flutter. Виджеты в стиле Купертино. CupertinoPageTransition. CupertinoPageScaffold. CupertinoNavigationBar
26. Flutter. Виджеты в стиле Купертино. CupertinoDialog. CupertinoButton. CupertinoContextMenu. CupertinoDatePicker. CupertinoTimerPicker.
27. Flutter. Виджеты в стиле Купертино. CupertinoTabScaffold. CupertinoTabView. CupertinoTextField. CupertinoButton.
28. Flutter. Навигация. PageView.
29. Flutter. Platform channel. Процесс создания. И способы его применения.
30. Flutter. File System. Чтение информации из файла.
31. Flutter. Sqflite.Управление базой данных. SharedPreference.
32. Flutter. Permissions. Notifications
33. Flutter. Firebase. Сервисы Firebase.
34. Flutter. FutureBuilder. StreamBuilder.
35. BLoC. Преимущества и недостатки. Схема.
36. Provider. Примеры использования.
37. ORM Hive. Преимущества и недостатки. Способ хранения данных.
38. Flutter Unit Tests. Mockito. Использование Mockito в написании тестов.
39. Flutter Widget Tests. Поиск Widget’ов.
40. Flutter Integration Tests. Запуск интеграционных тестов.

**1. Кроссплатформенная разработка мобильных приложений. Плюсы и минусы.**

**Кроссплатформенная разработка** — это подход, при котором одно приложение создаётся для нескольких платформ (iOS и Android) с использованием одного кода. Например, Flutter или React Native.

**Плюсы:**

* **Экономия времени и денег**: Один код вместо двух.
* **Упрощённое обслуживание**: Легче вносить изменения, так как код общий.
* **Быстрое прототипирование**: Подходит для создания MVP.

**Минусы:**

* **Меньшая производительность**: В некоторых случаях медленнее, чем нативные приложения.
* **Ограниченный доступ к платформенным API**: Не всегда поддерживаются функции платформ.
* **Размер приложения**: Кроссплатформенные приложения часто больше по размеру.

**2. Dart. Преимущества, недостатки и платформы.**

**Преимущества:**

* **Простота изучения**: Лёгкий синтаксис.
* **Асинхронность**: Удобно работать с потоками данных и будущими задачами.
* **Широкая поддержка Flutter**: Основной язык для разработки на Flutter.

**Недостатки:**

* **Менее популярный**: Меньше библиотек и сообщества.
* **Ограниченность за пределами Flutter**: Dart редко используется вне экосистемы Flutter.

**Платформы:**

* Web (через компиляцию в JavaScript).
* Desktop (Windows, macOS, Linux).
* Mobile (iOS и Android через Flutter).
* Сервер (через dart:io).

**3. Dart Types. Основные типы.**

**Основные типы:**

1. **Numbers**: Числа (целые и дробные).
2. int a = 5;
3. double b = 3.14;
4. **Strings**: Текстовые строки.
5. String greeting = 'Hello, Dart!';
6. **Booleans**: Логические значения.
7. bool isActive = true;
8. **Lists**: Списки (массивы).
9. List<int> numbers = [1, 2, 3];
10. **Sets**: Наборы уникальных значений.
11. Set<String> fruits = {'apple', 'banana'};
12. **Maps**: Ключ-значение.
13. Map<String, int> scores = {'Alice': 90, 'Bob': 85};
14. **Runes and Grapheme Clusters**: Работа с Unicode-символами.
15. **Symbols**: Для идентификаторов.
16. Symbol sym = #mySymbol;

**4. Dart Class. Структура класса, наследование, интерфейсы.**

**Пример класса:**

class Animal {

String name;

Animal(this.name);

void speak() {

print('$name makes a noise');

}

}

**Наследование:**

class Dog extends Animal {

Dog(String name) : super(name);

@override

void speak() {

print('$name barks');

}

}

**Интерфейс:**

В Dart любой класс можно использовать как интерфейс.

abstract class Flyable {

void fly();

}

class Bird implements Flyable {

@override

void fly() {

print('Flying!');

}

}

в чем отличие класса от интерфейса?

Отличие **класса** от **интерфейса** заключается в их назначении и способах использования:

### ****Класс****

* **Назначение**: Класс используется для описания объекта с конкретной реализацией методов и свойств.
* **Реализация**: Может содержать как реализацию методов, так и свойства.
* **Наследование**: Один класс может наследовать только один другой класс (одиночное наследование).
* **Пример**:
* class Animal {
* void eat() {
* print("Animal is eating");
* }
* }

### ****Интерфейс****

* **Назначение**: Интерфейс определяет **контракт**, который должен быть реализован классом. Интерфейс не содержит реализации методов (только их сигнатуры).
* **Реализация**: Класс, который реализует интерфейс, обязан реализовать все методы интерфейса.
* **Использование в Dart**: В Dart любой класс может быть использован как интерфейс.
* **Множественная реализация**: Класс может реализовывать несколько интерфейсов одновременно.
* **Пример**:
* abstract class Flyable {
* void fly(); // Метод без реализации
* }
* class Bird implements Flyable {
* @override
* void fly() {
* print("Bird is flying");
* }
* }

### ****Ключевые различия****

| **Свойство** | **Класс** | **Интерфейс** |
| --- | --- | --- |
| **Цель** | Создание объектов с конкретной реализацией. | Определение обязательных методов/свойств. |
| **Методы** | Может содержать реализацию методов. | Методы не имеют реализации (в abstract). |
| **Использование** | Для моделирования объектов. | Для установления контракта. |
| **Наследование/реализация** | Только один базовый класс. | Можно реализовать несколько интерфейсов. |

### ****Когда использовать?****

* **Класс**: Когда нужно создать объект с уже определённым поведением.
* **Интерфейс**: Когда нужно задать обязательный набор методов/свойств для различных классов, не заботясь о реализации.

**5. Dart Mixins. Примеси и интерфейсы.**

* **Примеси (Mixins)** позволяют добавлять функционал без наследования.
* **Интерфейсы** — это только контракт, примеси — это реальная реализация.

mixin Walkable {

void walk() {

print('Walking');

}

}

class Human with Walkable {}

**6. Dart Functions. Типы и параметризация.**

**Пример функции:**

int add(int a, int b) => a + b;

**Типы параметров:**

* **Позиционные**:
* void greet(String name, [String? title]) {
* print('Hello $title $name');
* }
* **Именованные**:
* void greet({required String name, String? title}) {
* print('Hello $title $name');
* }

какие типы функций бывают и расскажи подробрнее разницу позиционных и именованных

### ****Типы функций в Dart****

Функции в Dart можно разделить на несколько типов:

1. **Обычные функции**  
   Функции, принимающие параметры и возвращающие значение.
2. int add(int a, int b) {
3. return a + b;
4. }
5. **Анонимные функции (ламбды)**  
   Функции без имени, используемые для краткости.
6. var multiply = (int a, int b) => a \* b;
7. **Асинхронные функции**  
   Возвращают Future и используются для работы с асинхронным кодом.
8. Future<String> fetchData() async {
9. return "Data fetched";
10. }
11. **Геттеры и сеттеры**  
    Функции, работающие с приватными полями класса.
12. class Person {
13. String \_name;
14. String get name => \_name; // Геттер
15. set name(String value) => \_name = value; // Сеттер
16. }
17. **Рекурсивные функции**  
    Функции, которые вызывают сами себя.
18. int factorial(int n) {
19. if (n == 0) return 1;
20. return n \* factorial(n - 1);
21. }
22. **Статические функции**  
    Функции, привязанные к классу, но не к его экземпляру.
23. class MathUtils {
24. static int square(int n) => n \* n;
25. }

### ****Позиционные и именованные параметры****

#### ****1. Позиционные параметры****

* **Описание**: Параметры передаются по порядку. Если параметр не передан, используется значение по умолчанию (если указано).
* **Синтаксис**: Используются квадратные скобки [] для обозначения необязательных параметров.
* **Пример**:
* void greet(String name, [String? message = "Hello"]) {
* print("$message, $name!");
* }
* greet("Alice"); // Вывод: Hello, Alice!
* greet("Bob", "Hi"); // Вывод: Hi, Bob!
* **Плюсы**: Легко читаются, используются для небольшого числа параметров.
* **Минусы**: Если у функции много параметров, порядок их вызова может быть неудобным и путаным.

#### ****2. Именованные параметры****

* **Описание**: Параметры передаются в произвольном порядке, указывая их имя.
* **Синтаксис**: Используются фигурные скобки {}. Для обозначения обязательных именованных параметров добавляется required.
* **Пример**:
* void greet({required String name, String message = "Hello"}) {
* print("$message, $name!");
* }
* greet(name: "Alice"); // Вывод: Hello, Alice!
* greet(name: "Bob", message: "Hi"); // Вывод: Hi, Bob!
* **Плюсы**: Удобно для функций с большим числом параметров, повышает читаемость кода.
* **Минусы**: Требует явного указания имен параметров.

### ****Сравнение позиционных и именованных параметров****

| **Свойство** | **Позиционные параметры** | **Именованные параметры** |
| --- | --- | --- |
| **Порядок передачи** | Важен порядок параметров. | Порядок передачи не имеет значения. |
| **Читаемость** | Менее читаемые при большом количестве. | Лучше читаются, особенно в сложных функциях. |
| **Обязательность** | Используется по умолчанию или передаётся. | Можно указать через required. |
| **Удобство использования** | Хорошо подходит для коротких функций. | Подходит для сложных функций. |

### ****Когда использовать?****

* **Позиционные**: Если параметров мало и их порядок очевиден.
* **Именованные**: Если параметров много или некоторые из них необязательные.

**7. Dart Future.**

* **Future** используется для асинхронного выполнения задач.
* Пример:
* Future<String> fetchData() async {
* await Future.delayed(Duration(seconds: 2));
* return 'Data loaded';
* }

**8. Dart Stream.**

* **Stream** передаёт данные последовательно.
* Пример:
* Stream<int> countStream() async\* {
* for (int i = 1; i <= 5; i++) {
* yield i;
* }
* }

**9. Dart Isolate.**

* **Isolate** позволяет выполнять код в отдельных потоках.
* Пример:
* import 'dart:isolate';
* void printMessage(String message) {
* print(message);
* }
* void main() {
* Isolate.spawn(printMessage, 'Hello from Isolate!');
* }

**10. Flutter. Свойства и архитектура.**

**Свойства:**

* **Кроссплатформенность**: Одна кодовая база для iOS и Android.
* **Высокая производительность**: Использует движок C++ (Skia).
* **Гибкость**: Виджеты для всего.

**Архитектура:**

* **UI Layer**: Виджеты.
* **Framework Layer**: Логика (Dart).
* **Engine Layer**: Рендеринг (Skia).

расскажи подробнее про архитектуру flutter

### ****Архитектура Flutter****

Flutter — это UI-фреймворк от Google, который позволяет создавать нативные приложения для iOS, Android, веба и настольных платформ. Его архитектура устроена так, чтобы предоставлять высокую производительность и гибкость для создания пользовательских интерфейсов.

### ****Ключевые компоненты архитектуры Flutter****

#### ****1. Flutter Framework****

* **Описание**: Это верхний уровень архитектуры, написанный на Dart. Здесь находятся все основные компоненты для разработки приложений:
  + **UI компоненты**: Виджеты, работающие как строительные блоки.
  + **Гибкость**: Возможность кастомизировать интерфейсы на любом уровне.
* **Составные части:**
  + **Widgets Layer**: Основной слой, содержащий все виджеты для создания интерфейсов.
  + **Rendering Layer**: Отвечает за построение элементов интерфейса и их рендеринг.
  + **Animation and Gestures**: Поддержка анимаций и жестов.

#### ****2. Flutter Engine****

* **Описание**: Это низкоуровневый компонент, написанный на C++. Он управляет рендерингом и взаимодействует с платформенными API.
* **Функциональность**:
  + Рендеринг с использованием **Skia Graphics Engine** для высокопроизводительных отрисовок.
  + Работа с текстом, шрифтами и анимациями.
  + Обработка жестов и событий.

#### ****3. Platform-Specific Embedders****

* **Описание**: Этот слой обеспечивает связь между Flutter и нативными платформами (Android, iOS, веб и т. д.).
* **Функциональность**:
  + Работа с нативными компонентами (например, GPS, камера, доступ к файловой системе).
  + Интеграция через платформенные каналы (**Platform Channels**).

### ****Архитектурный поток данных****

1. **Виджеты (Widgets)**:
   * Начальный слой приложения.
   * Строится как дерево (Widget Tree).
   * Примеры: Text, Container, Row, Column.
2. **Слой рендера (Rendering)**:
   * Преобразует дерево виджетов в **Element Tree**.
   * Создаёт **Render Tree** для отрисовки интерфейса.
3. **Реактивная архитектура**:
   * Flutter использует однонаправленный поток данных:
     + Изменение состояния (State) вызывает перестроение соответствующих виджетов.
   * Пример:
   * setState(() {
   * counter++;
   * });
4. **Компиляция**:
   * **JIT (Just-In-Time)** во время разработки:
     + Поддерживает "горячую перезагрузку" для мгновенного отображения изменений.
   * **AOT (Ahead-Of-Time)** при публикации:
     + Обеспечивает высокую производительность.

### ****Преимущества архитектуры Flutter****

1. **Единая кодовая база**:
   * Написание одного кода для всех платформ.
2. **Высокая производительность**:
   * Использование **Skia** и нативного кода для обеспечения 60–120 FPS.
3. **Горячая перезагрузка**:
   * Позволяет разработчикам мгновенно видеть изменения без перезапуска приложения.
4. **Кастомизация интерфейса**:
   * Легко создавать сложные UI-компоненты без ограничения платформенными стандартами.

### ****Пример взаимодействия компонентов****

Простое приложение на Flutter: счётчик с кнопкой увеличения.

1. **Flutter Framework**:
   * Виджет Scaffold создаёт основную структуру.
   * Виджет Text отображает число.
   * Виджет FloatingActionButton обрабатывает нажатия.
2. **Flutter Engine**:
   * Рендерит обновлённый текст и кнопку.
3. **Platform-Specific Embedders**:
   * Поддерживает платформенные функции, такие как жесты или системные темы.

### ****Вывод****

Архитектура Flutter построена так, чтобы упростить создание интерфейсов с высокой производительностью, сохраняя при этом гибкость. Она объединяет в себе мощные виджеты, эффективный движок рендеринга и поддержку нативных функций через платформенные каналы.

### ****11. Flutter. Структура проекта****

Flutter-проект имеет следующую структуру:

project\_name/

├── android/ # Платформоспецифичные файлы для Android.

├── ios/ # Платформоспецифичные файлы для iOS.

├── lib/ # Основная папка, где находится код приложения.

│ ├── main.dart # Главный файл приложения.

│ ├── screens/ # Экраны приложения.

│ ├── widgets/ # Повторно используемые виджеты.

│ └── utils/ # Утилиты, вспомогательные функции.

├── test/ # Тесты.

├── web/ # Платформоспецифичные файлы для веба.

├── pubspec.yaml # Файл конфигурации проекта (зависимости, ресурсы).

**Пример**: Файл main.dart:

import 'package:flutter/material.dart';

void main() {

runApp(MyApp());

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text("Hello Flutter")),

body: Center(child: Text("Welcome to Flutter")),

),

);

}

}

### ****12. Flutter Widget. Flutter State. Типы виджетов****

* **Widget**: Это строительные блоки Flutter-приложений, которые описывают, как должен выглядеть интерфейс.
* **State**: Это данные, которые могут изменяться во время работы приложения.

**Типы виджетов**:

1. **StatelessWidget**: Виджеты без изменяемого состояния (например, статический текст).
2. **StatefulWidget**: Виджеты с изменяемым состоянием (например, кнопка-счётчик).

**Пример**:

class MyStatelessWidget extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text("I am stateless");

}

}

### ****13. StatelessWidget vs StatefulWidget****

* **StatelessWidget**:
  + Не имеет изменяемого состояния.
  + Подходит для статических элементов интерфейса.
* class MyStateless extends StatelessWidget {
* @override
* Widget build(BuildContext context) {
* return Text("Static text");
* }
* }
* **StatefulWidget**:
  + Имеет состояние, которое может изменяться с помощью setState.
  + Используется для динамических элементов.
* class MyStateful extends StatefulWidget {
* @override
* \_MyStatefulState createState() => \_MyStatefulState();
* }
* class \_MyStatefulState extends State<MyStateful> {
* int counter = 0;
* @override
* Widget build(BuildContext context) {
* return Column(
* children: [
* Text("Counter: $counter"),
* ElevatedButton(
* onPressed: () {
* setState(() {
* counter++;
* });
* },
* child: Text("Increment"),
* )
* ],
* );
* }
* }

### ****14. Flutter Widget Tree, Element Tree, Render Tree****

#### ****Widget Tree****

* Это иерархия всех виджетов, используемых в приложении.
* Описывает, как должен выглядеть пользовательский интерфейс.
* Создаётся во время выполнения программы и используется как шаблон для отображения интерфейса.

**Пример**:

Widget build(BuildContext context) {

return Column(

children: [

Text("Hello World"),

ElevatedButton(

onPressed: () {},

child: Text("Click Me"),

),

],

);

}

Этот код создаёт **Widget Tree**:

* Корневой виджет Column, который содержит два дочерних виджета: Text и ElevatedButton.

#### ****Element Tree****

* Отражает **экземпляры** виджетов во время выполнения.
* Указывает, какие виджеты активны в памяти.
* Обновляется при перестроении виджетов (rebuild).

#### ****Render Tree****

* Описывает, как элементы из **Widget Tree** должны быть отображены на экране.
* Создаётся из **Element Tree**.
* Содержит данные о размере, положении и рисовании элементов интерфейса.

### ****15. Layout Widgets (Single-child layout widgets)****

#### 1. ****Align****

* Выравнивает дочерний виджет в пределах контейнера.
* Параметр alignment указывает позицию (например, центр, левый край).

**Пример**:

Align(

alignment: Alignment.center,

child: Text("Centered Text"),

);

#### 2. ****AspectRatio****

* Задаёт соотношение ширины к высоте для дочернего виджета.

**Пример**:

AspectRatio(

aspectRatio: 16 / 9,

child: Container(color: Colors.blue),

);

#### 3. ****Baseline****

* Выравнивает дочерний виджет по базовой линии текста.

**Пример**:

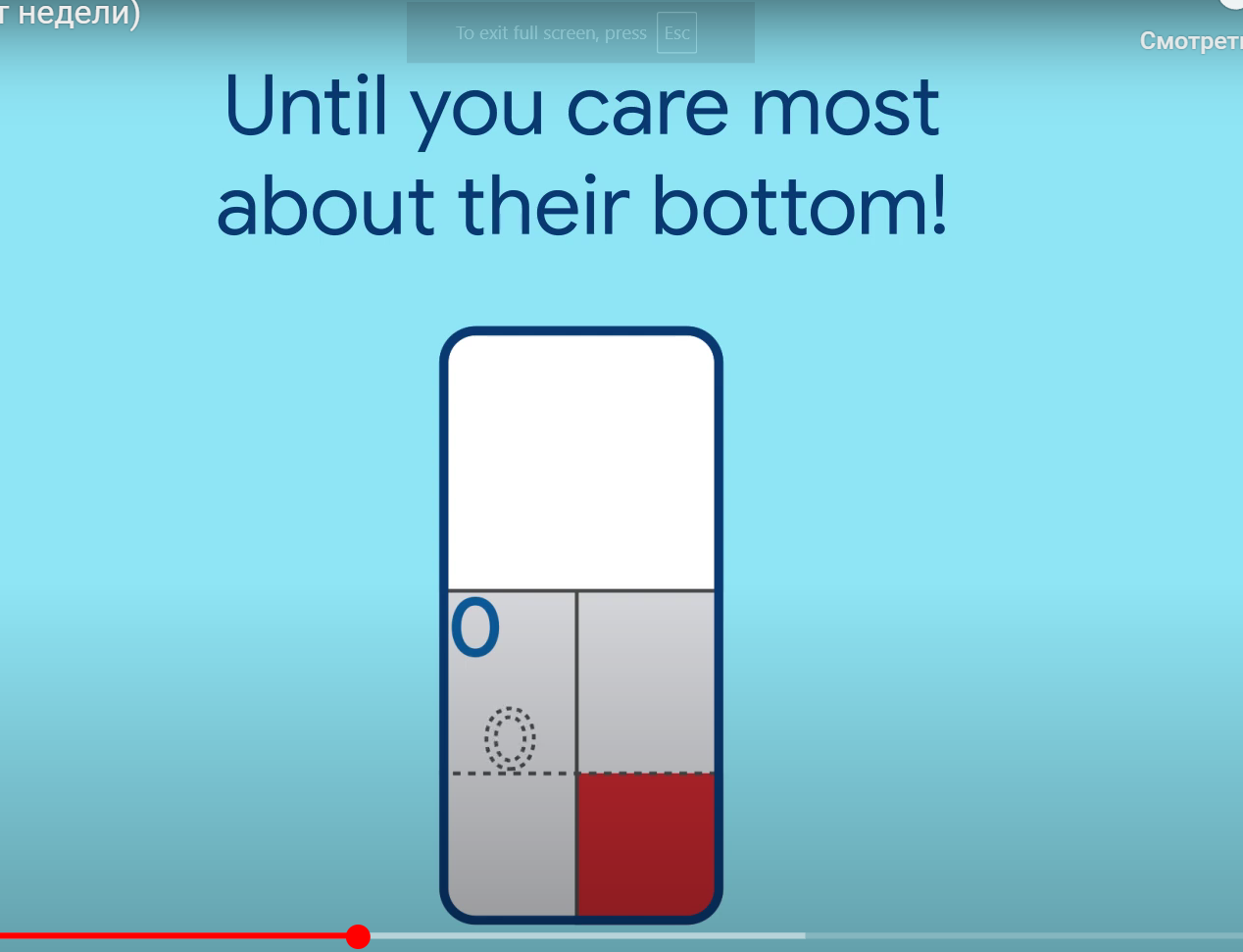
Baseline(

baseline: 50.0,

baselineType: TextBaseline.alphabetic,

child: Text("Baseline Alignment"),

);



#### 4. ****Center****

* Центрирует дочерний элемент.

**Пример**:

Center(

child: Text("I am centered"),

);

#### 5. ****ConstrainedBox****

* Ограничивает размер дочернего виджета.

**Пример**:

ConstrainedBox(

constraints: BoxConstraints(maxWidth: 100, maxHeight: 100),

child: Container(color: Colors.red),

);

#### 6. ****Container****

* Универсальный виджет для оформления, добавления отступов, цветов, размеров и границ.

**Пример**:

Container(

padding: EdgeInsets.all(10),

color: Colors.green,

child: Text("Container Example"),

);

#### 7. ****MediaQuery****

* Позволяет получать информацию о размере экрана, ориентации, плотности пикселей и т.д.

**Пример**:

MediaQuery.of(context).size.width;

### ****16. Layout Widgets (Single-child layout widgets)****

#### 1. ****Expanded****

* Расширяет дочерний виджет, занимая доступное пространство внутри контейнера.

**Пример**:

Row(

children: [

Expanded(child: Container(color: Colors.red)),

Container(width: 50, color: Colors.blue),

],

);

#### 2. ****Padding****

* Добавляет отступы вокруг дочернего виджета.

**Пример**:

Padding(

padding: EdgeInsets.all(16.0),

child: Text("Text with padding"),

);

#### 3. ****SizedBox****

* Устанавливает фиксированные размеры для дочернего элемента.

**Пример**:

SizedBox(

width: 100,

height: 50,

child: ElevatedButton(onPressed: () {}, child: Text("Button")),

);

#### 4. ****Transform****

* Преобразует дочерний виджет (вращение, масштабирование и т.д.).

**Пример**:

Transform.rotate(

angle: 0.1,

child: Text("Rotated Text"),

);

#### 5. ****OverflowBox****

* Позволяет дочернему виджету выходить за пределы родительских ограничений.

**Пример**:

OverflowBox(

maxWidth: 200,

child: Container(width: 300, height: 100, color: Colors.yellow),

);

#### 6. ****FittedBox****

* Масштабирует дочерний виджет, чтобы он поместился в контейнер.

**Пример**:

FittedBox(

child: Text("Scaled Text"),

);

### ****17. Layout Widgets (Multi-child layout widgets)****

#### 1. ****Column****

* Располагает дочерние виджеты вертикально.

**Пример**:

Column(

children: [

Text("Item 1"),

Text("Item 2"),

],

);

#### 2. ****GridView****

* Создаёт прокручиваемую сетку.

**Пример**:

GridView.count(

crossAxisCount: 2,

children: [Text("1"), Text("2"), Text("3")],

);

#### 3. ****Stack****

* Располагает виджеты друг над другом.

**Пример**:

Stack(

children: [

Container(color: Colors.red, width: 100, height: 100),

Text("Overlapping text"),

],

);

#### 4. ****IndexedStack****

* Показывает только один дочерний элемент, определяемый индексом.

**Пример**:

IndexedStack(

index: 0,

children: [

Text("First"),

Text("Second"),

],

);

### ****18. Layout Widgets (Multi-child layout widgets, продолжение)****

#### 1. ****Row****

* Располагает виджеты горизонтально.

**Пример**:

Row(

children: [

Text("Left"),

Text("Right"),

],

);

#### 2. ****ListView****

* Прокручиваемый список.

**Пример**:

ListView(

children: [Text("1"), Text("2")],

);

#### 3. ****ListTile****

* Удобный виджет для списков с заголовком, подзаголовком и иконкой.

**Пример**:

ListTile(

leading: Icon(Icons.person),

title: Text("Name"),

subtitle: Text("Details"),

);

#### 4. ****Table****

* Создаёт таблицу.

**Пример**:

Table(

children: [

TableRow(children: [Text("Cell 1"), Text("Cell 2")]),

],

);

#### 5. ****Wrap****

* Располагает элементы в несколько строк или столбцов.

**Пример**:

Wrap(

children: [

Text("Item 1"),

Text("Item 2"),

Text("Item 3"),

],

);

### ****19. UI Components****

#### 1. ****AppBar****

* Верхняя панель приложения.

**Пример**:

AppBar(

title: Text("App Bar Example"),

);

#### 2. ****RaisedButton****

* Кнопка с эффектом подъёма (устарел, заменён на ElevatedButton).

#### 3. ****Scaffold****

* Основная структура страницы.

**Пример**:

Scaffold(

appBar: AppBar(title: Text("Scaffold Example")),

body: Center(child: Text("Body Content")),

);

#### 4. ****FloatingActionButton****

* Плавающая кнопка для важных действий.

**Пример**:

FloatingActionButton(

onPressed: () {},

child: Icon(Icons.add),

);

### ****20. UI Components (Drawer, PopupButton, Date&Time Picker)****

#### 1. ****Drawer****

* Боковая панель для навигации.

**Пример**:

Drawer(

child: ListView(children: [ListTile(title: Text("Menu Item"))]),

);

#### 2. ****PopupMenuButton****

* Меню с вариантами выбора.

**Пример**:

PopupMenuButton(

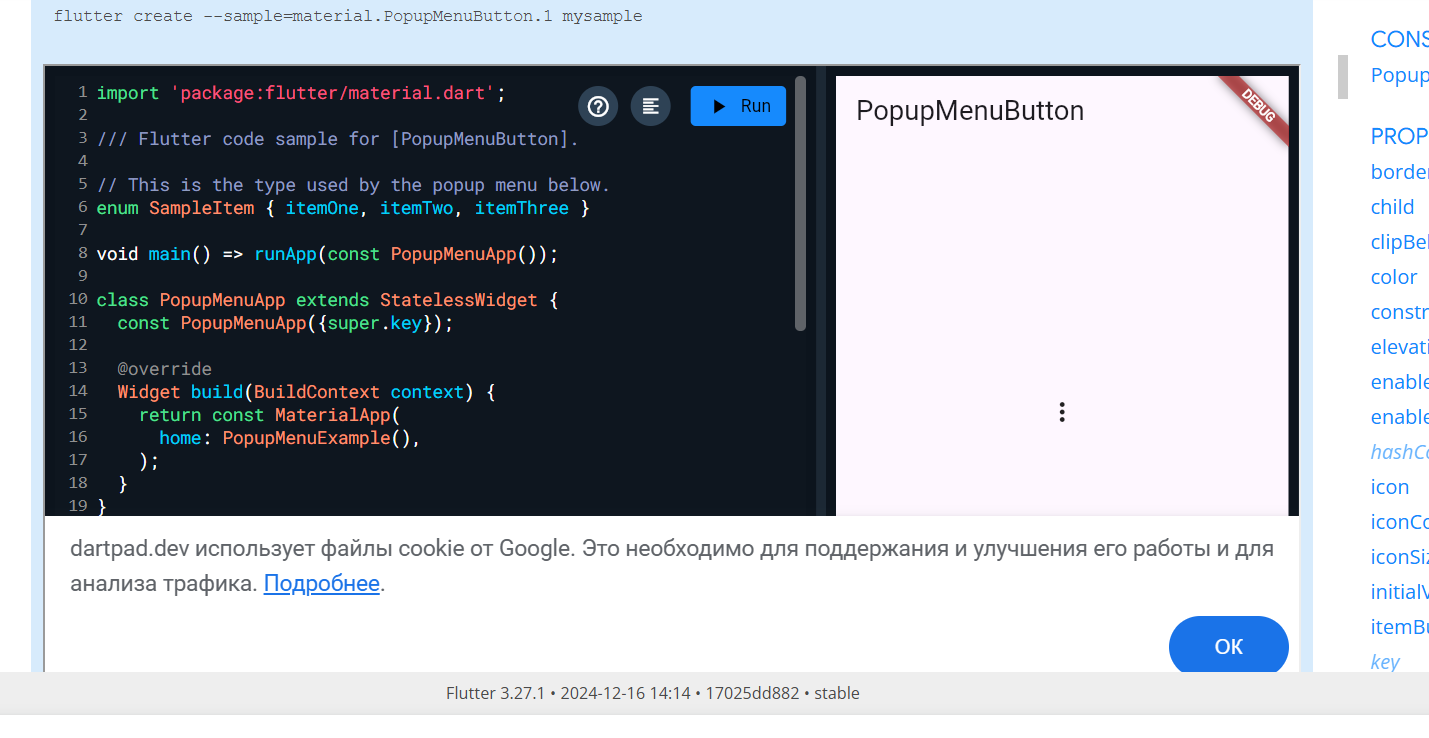
itemBuilder: (context) => [

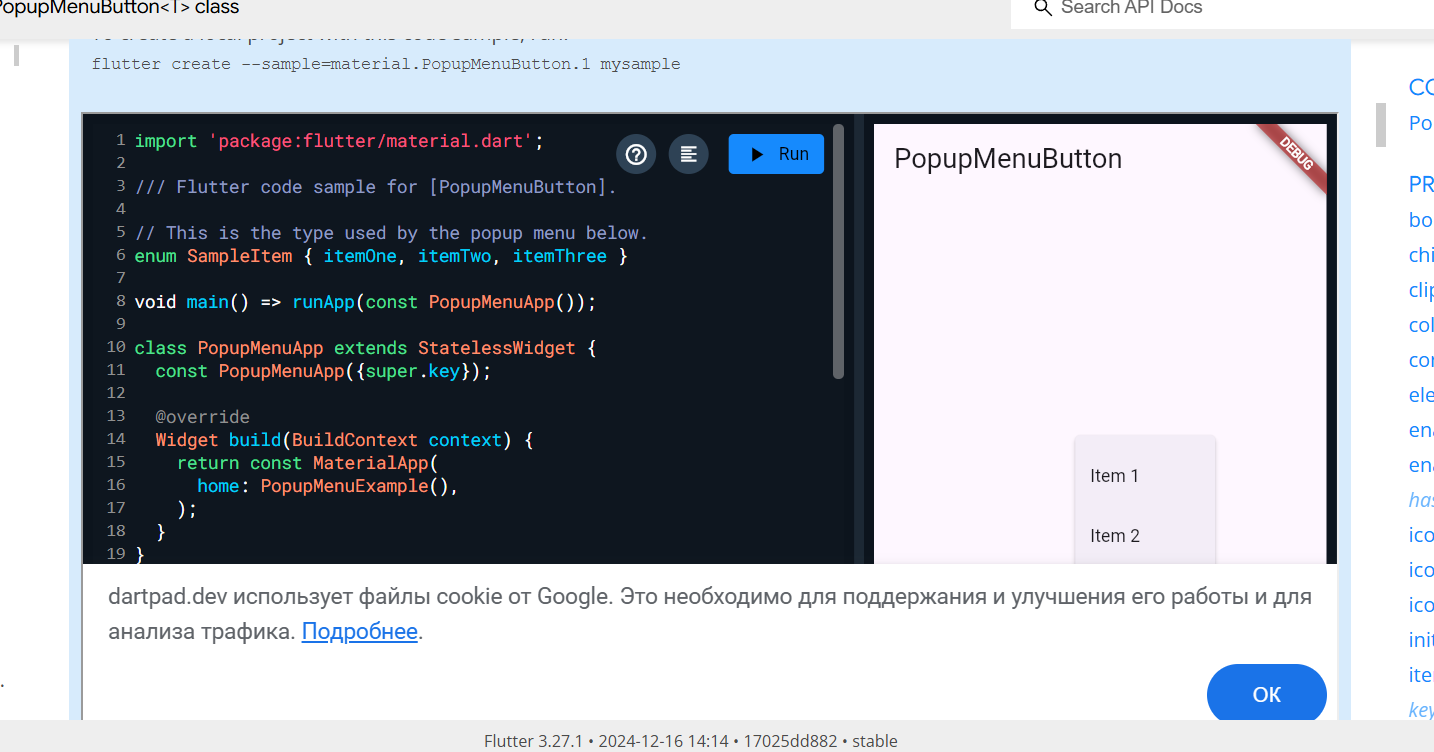
PopupMenuItem(child: Text("Option 1")),

PopupMenuItem(child: Text("Option 2")),

],

);





#### 3. ****Date & Time Picker****

* Виджеты для выбора даты и времени.

**Пример**:

showDatePicker(

context: context,

initialDate: DateTime.now(),

firstDate: DateTime(2000),

lastDate: DateTime(2100),

);

**21. Flutter. Компоненты пользовательского интерфейса**

**AlertDialog**

Модальное диалоговое окно для отображения предупреждений или сообщений пользователю.  
**Пример:**

showDialog(

context: context,

builder: (context) => AlertDialog(

title: Text('Warning'),

content: Text('This is an alert dialog.'),

actions: [

TextButton(

onPressed: () => Navigator.pop(context),

child: Text('Cancel'),

),

TextButton(

onPressed: () => Navigator.pop(context),

child: Text('OK'),

),

],

),

);

**BottomSheet**

Всплывающее окно внизу экрана. Может быть модальным (showModalBottomSheet) или постоянным (BottomSheet).  
**Пример:**

showModalBottomSheet(

context: context,

builder: (context) => Container(

height: 200,

child: Center(child: Text('This is a bottom sheet')),

),

);

**ExpansionPanel**

Секция, которая раскрывается для отображения дополнительной информации. Обычно используется внутри ExpansionPanelList.  
**Пример:**

ExpansionPanelList(

expansionCallback: (int index, bool isExpanded) {

setState(() {

items[index].isExpanded = !isExpanded;

});

},

children: items.map((item) {

return ExpansionPanel(

headerBuilder: (context, isExpanded) => ListTile(

title: Text(item.header),

),

body: ListTile(

title: Text(item.body),

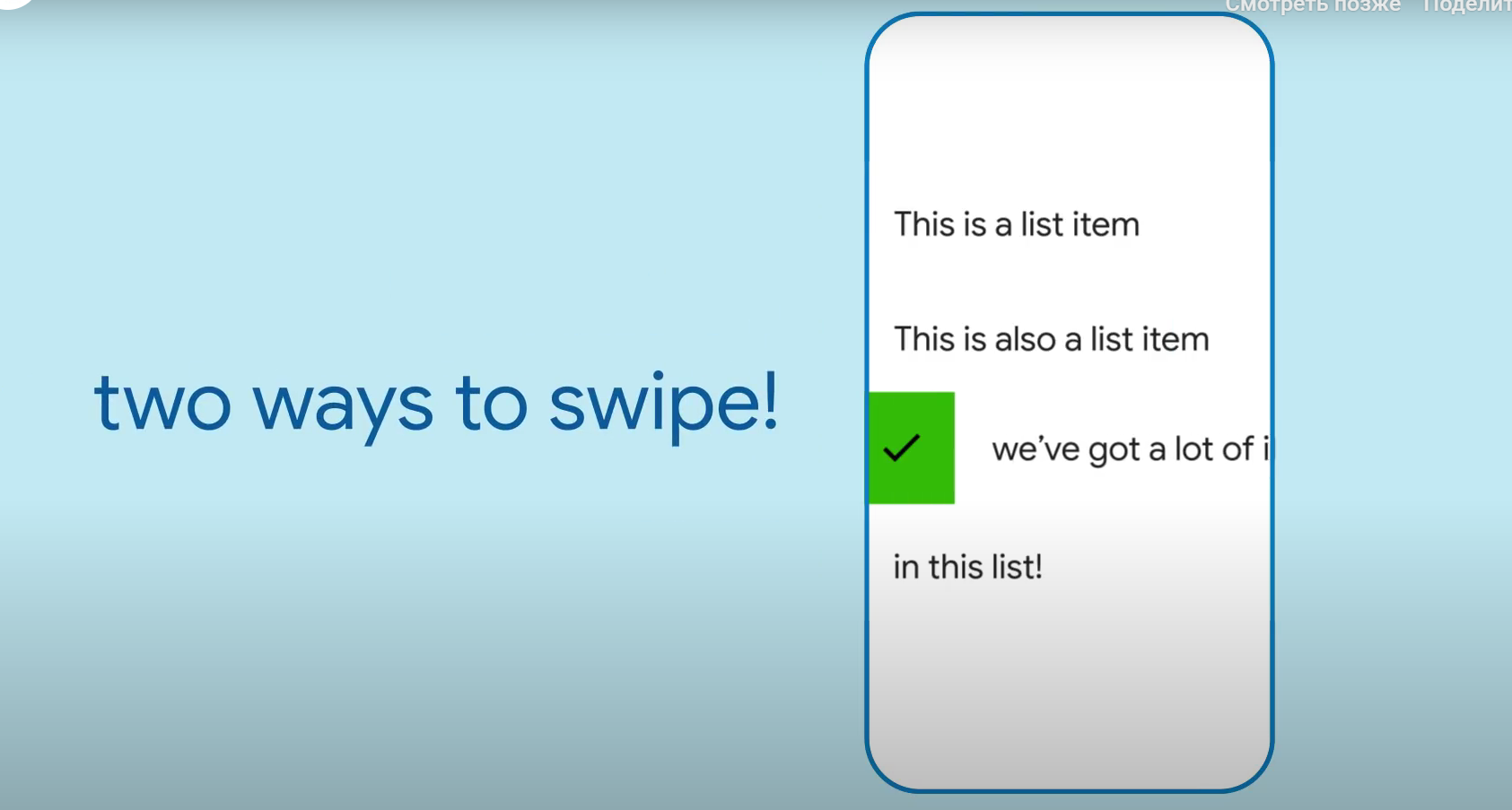
),

isExpanded: item.isExpanded,

);

}).toList(),

);



**SimpleDialog**

Простое модальное диалоговое окно для выбора из списка.  
**Пример:**

showDialog(

context: context,

builder: (context) => SimpleDialog(

title: Text('Choose an option'),

children: [

SimpleDialogOption(

onPressed: () => Navigator.pop(context, 'Option 1'),

child: Text('Option 1'),

),

SimpleDialogOption(

onPressed: () => Navigator.pop(context, 'Option 2'),

child: Text('Option 2'),

),

],

),

);

**SnackBar**

Временное уведомление внизу экрана.  
**Пример:**

ScaffoldMessenger.of(context).showSnackBar(

SnackBar(

content: Text('This is a SnackBar'),

action: SnackBarAction(

label: 'Undo',

onPressed: () {

// Действие при нажатии

},

),

),

);

**22. Flutter. Компоненты пользовательского интерфейса**

**Card**

Контейнер с закругленными углами и тенью.  
**Пример:**

Card(

elevation: 4,

child: Padding(

padding: EdgeInsets.all(16),

child: Text('This is a card'),

),

);

**Divider**

Горизонтальная линия для разделения элементов.  
**Пример:**

Divider(

thickness: 2,

color: Colors.grey,

);

**Stepper**

Виджет для пошагового отображения процесса.  
**Пример:**

Stepper(

steps: [

Step(title: Text('Step 1'), content: Text('Content for Step 1')),

Step(title: Text('Step 2'), content: Text('Content for Step 2')),

],

currentStep: \_currentStep,

onStepContinue: () {

if (\_currentStep < 2) {

setState(() => \_currentStep += 1);

}

},

onStepCancel: () {

if (\_currentStep > 0) {

setState(() => \_currentStep -= 1);

}

},

);

**Chip**

Компактный элемент для отображения информации, часто с аватаром или кнопкой действия.  
**Пример:**

Chip(

label: Text('Flutter'),

avatar: CircleAvatar(

child: Text('F'),

),

);

**23. Flutter Touch interactions**

**Dismissible**

Позволяет удалять элементы, проводя их влево или вправо.  
**Пример:**

Dismissible(

key: Key('item'),

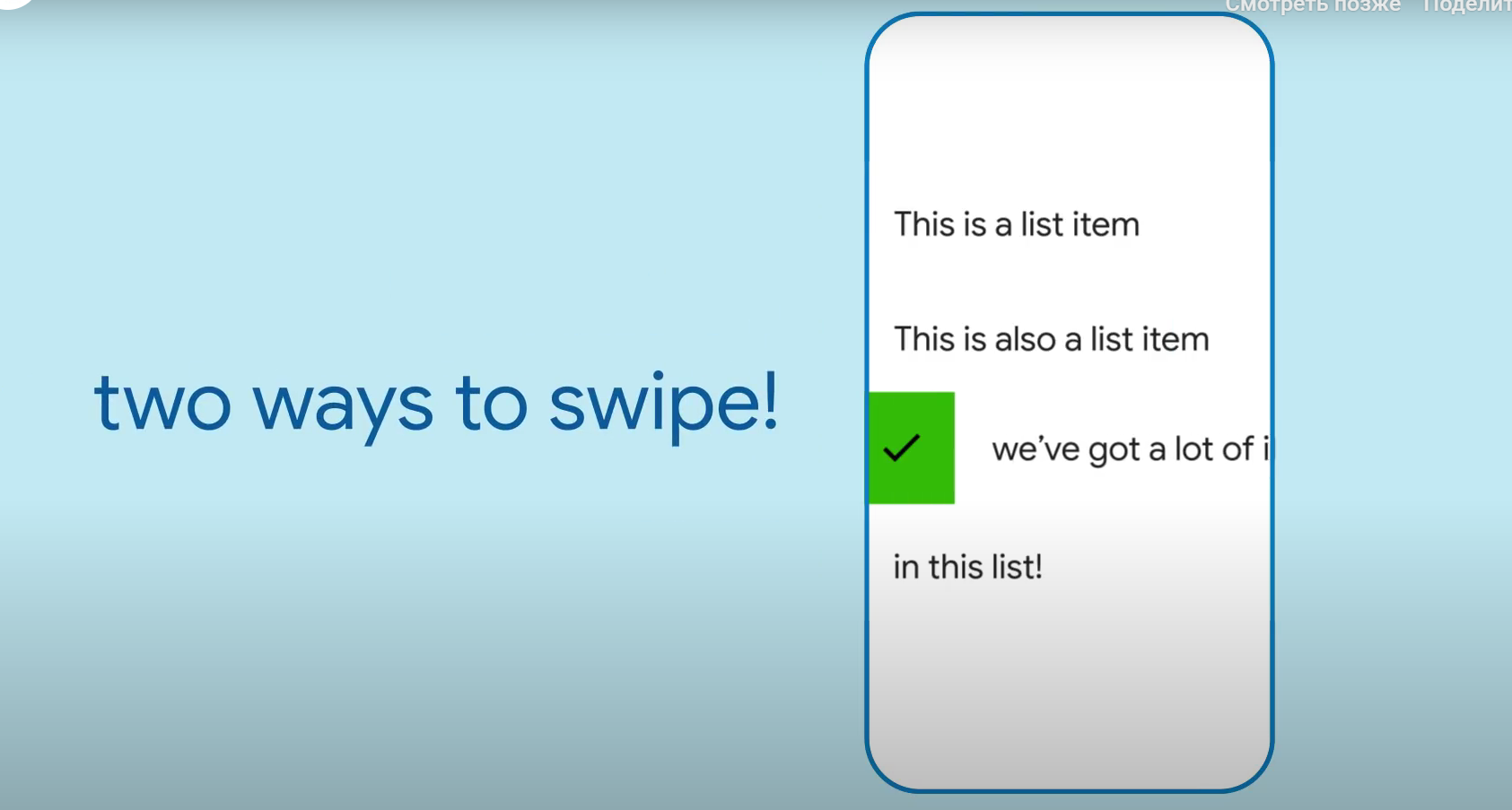
onDismissed: (direction) {

// Действие при удалении

},

child: ListTile(title: Text('Swipe to dismiss')),

);



**DragTarget**

Цель для перетаскиваемых объектов.  
**Пример:**

DragTarget<int>(

builder: (context, candidateData, rejectedData) => Container(

height: 100,

width: 100,

color: Colors.blue,

child: Center(child: Text('Drop here')),

),

onAccept: (data) {

// Действие при приеме

},

);

**Draggable**

Элемент, который можно перетаскивать.  
**Пример:**

Draggable<int>(

data: 1,

feedback: Container(

color: Colors.blue,

height: 50,

width: 50,

),

child: Container(

height: 50,

width: 50,

color: Colors.red,

),

);

**GestureDetector**

Обработка жестов, таких как нажатия, свайпы, двойные нажатия.  
**Пример:**

GestureDetector(

onTap: () {

// Действие при нажатии

},

child: Container(color: Colors.green),

);

**LongPressDraggable**

Элемент, который можно начать перетаскивать только после длительного нажатия.  
**Пример:**

LongPressDraggable<int>(

data: 1,

feedback: Container(color: Colors.blue, height: 50, width: 50),

child: Container(color: Colors.red, height: 50, width: 50),

);

### ****24. Flutter. Виджеты в стиле Купертино****

#### CupertinoActionSheet

Модальное окно внизу экрана, используемое для выбора из нескольких вариантов.  
**Пример:**

showCupertinoModalPopup(

context: context,

builder: (context) => CupertinoActionSheet(

title: Text('Choose an option'),

actions: [

CupertinoActionSheetAction(

onPressed: () => Navigator.pop(context),

child: Text('Option 1'),

),

CupertinoActionSheetAction(

onPressed: () => Navigator.pop(context),

child: Text('Option 2'),

),

],

cancelButton: CupertinoActionSheetAction(

onPressed: () => Navigator.pop(context),

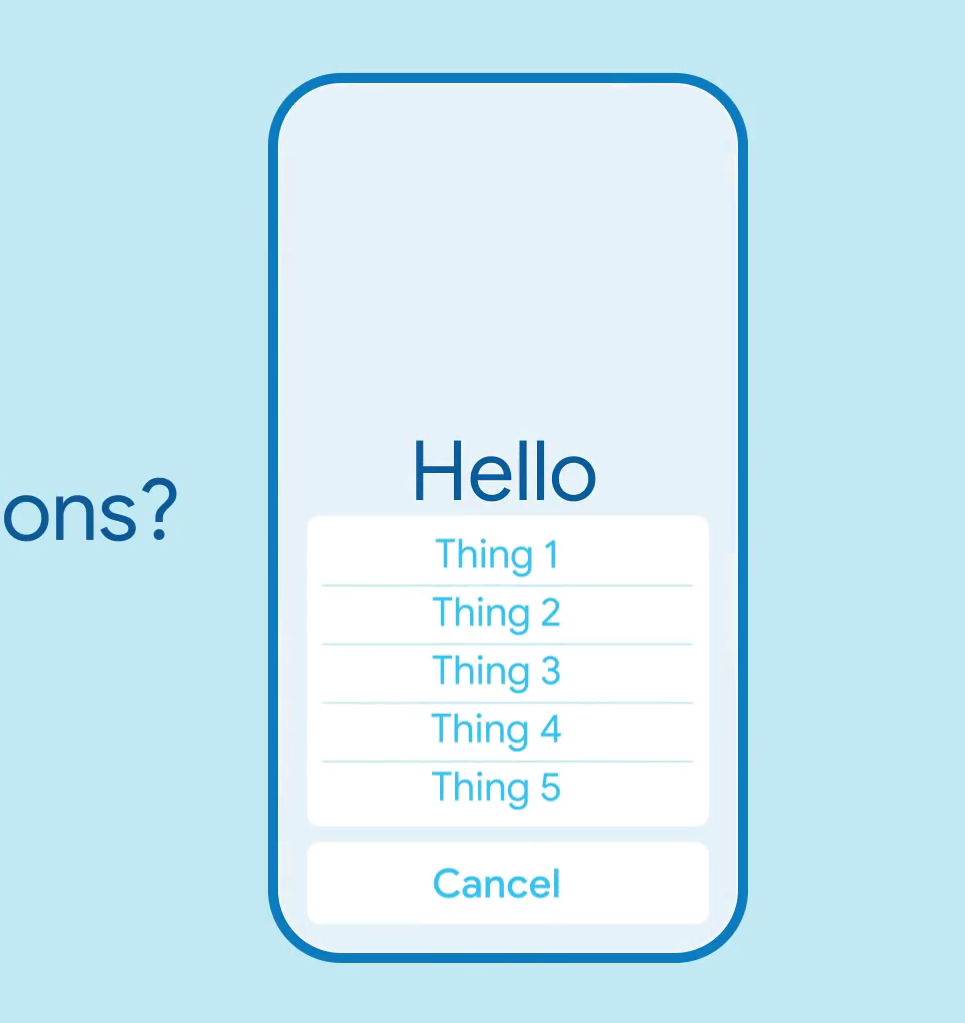
child: Text('Cancel'),

),

),

);





#### CupertinoSegmentedControl

Переключатель между несколькими параметрами, стилизованный под iOS.  
**Пример:**

CupertinoSegmentedControl<int>(

children: {

0: Text('Option 1'),

1: Text('Option 2'),

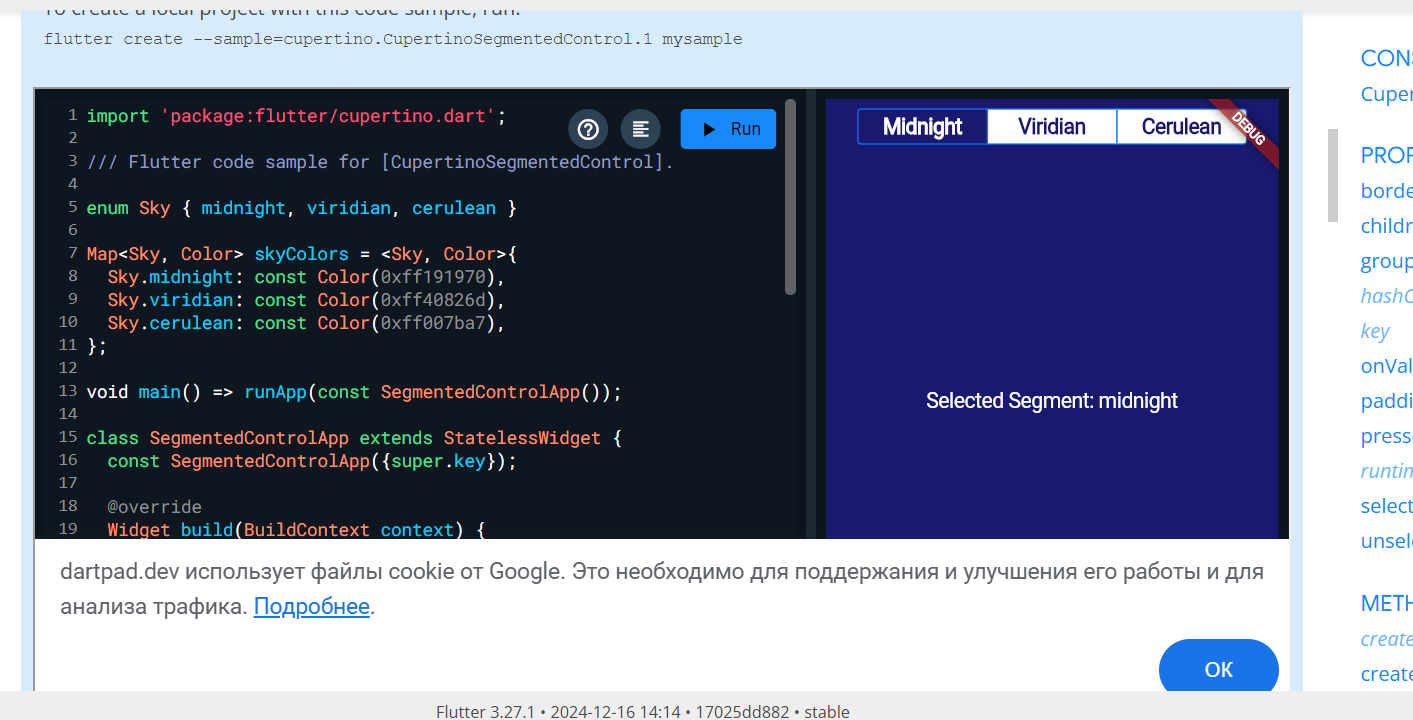
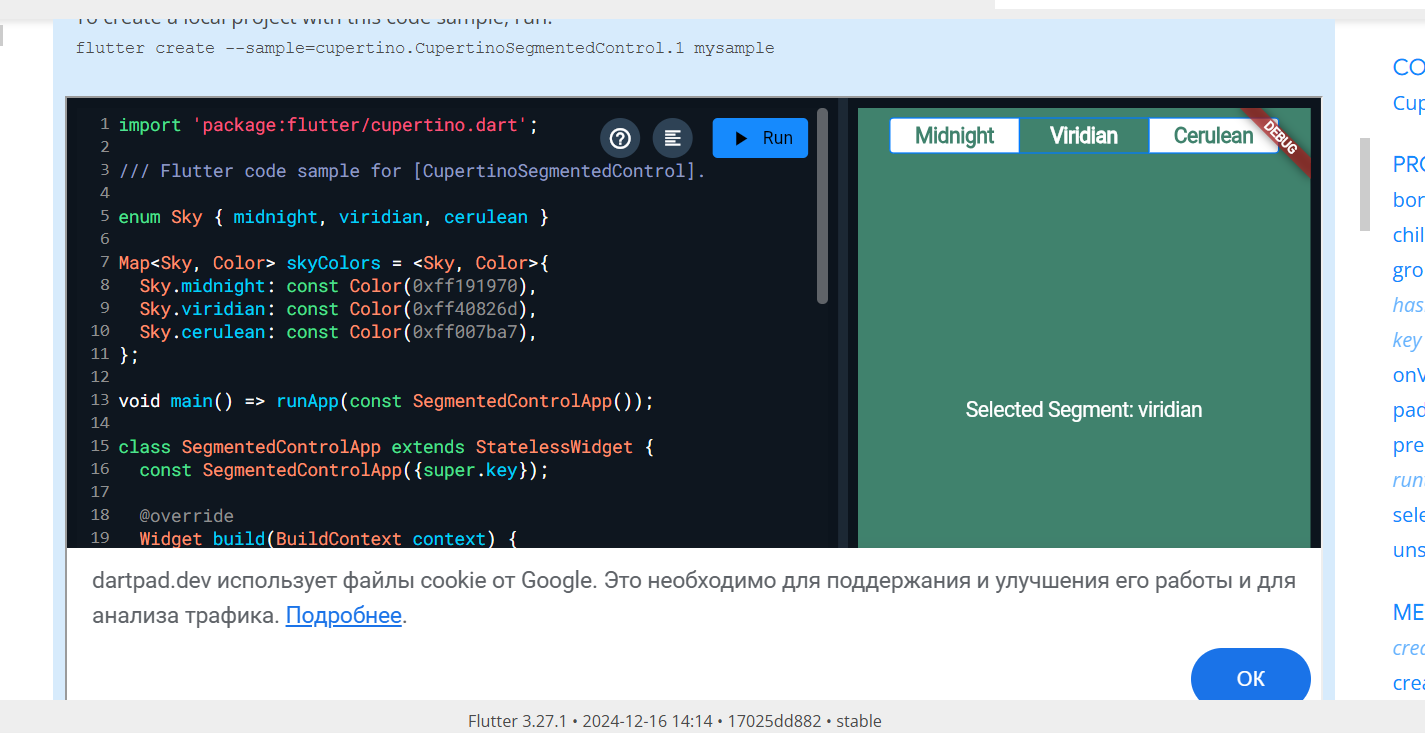
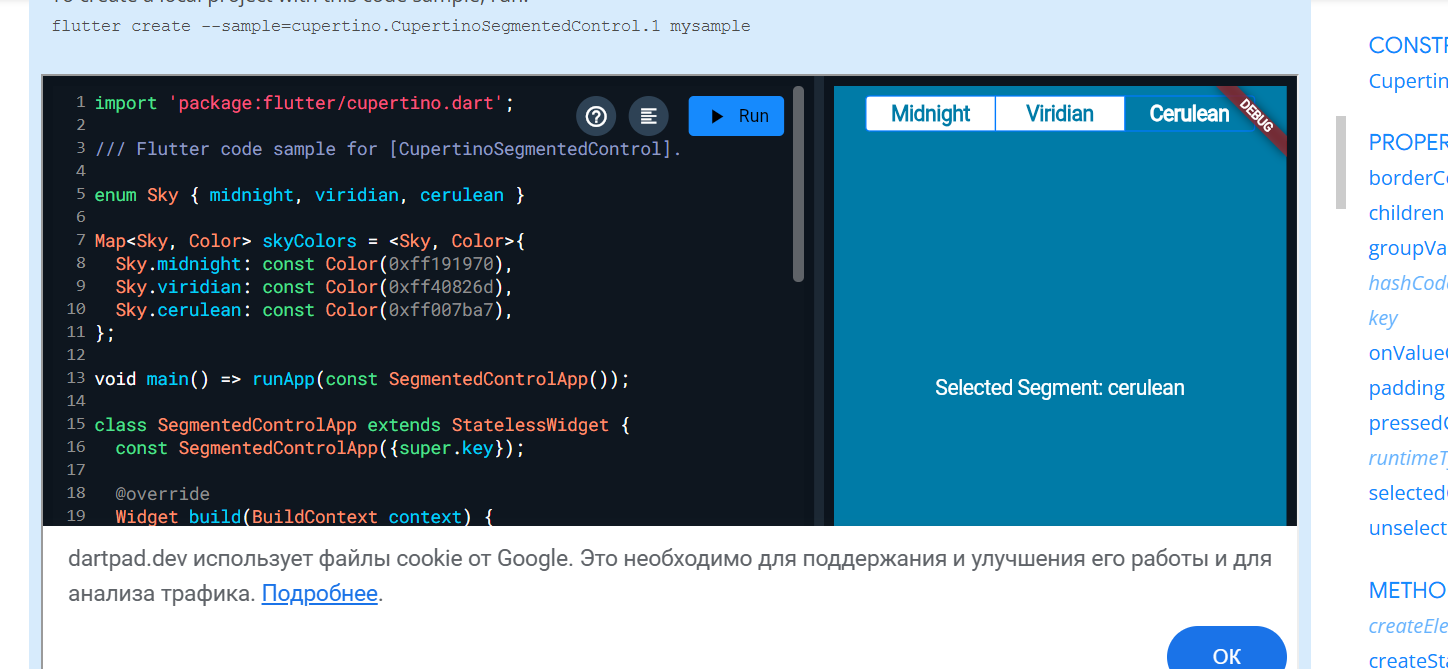
},

onValueChanged: (value) {

// Обработка выбора

},

);

#### CupertinoPicker

Колесо выбора, как в iOS.  
**Пример:**

CupertinoPicker(

itemExtent: 30.0,

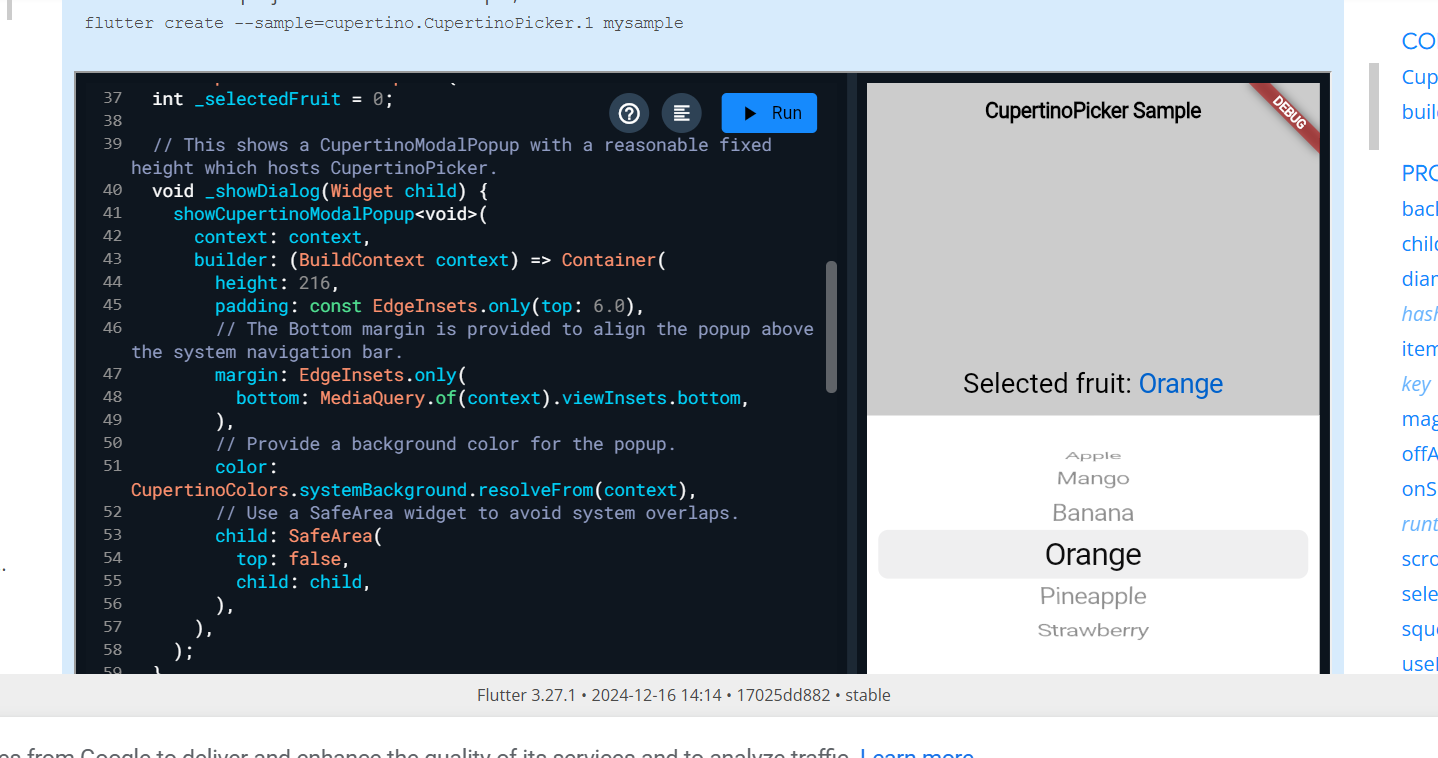
onSelectedItemChanged: (index) {

// Обработка выбора

},

children: List.generate(10, (index) => Text('Item $index')),

);



### ****25. Flutter. Виджеты в стиле Купертино****

#### CupertinoPageTransition

Анимация перехода между страницами, стилизованная под iOS.  
Используется внутри PageRoute.

#### CupertinoPageScaffold

Основной макет для страницы в стиле iOS.  
**Пример:**

CupertinoPageScaffold(

navigationBar: CupertinoNavigationBar(

middle: Text('Title'),

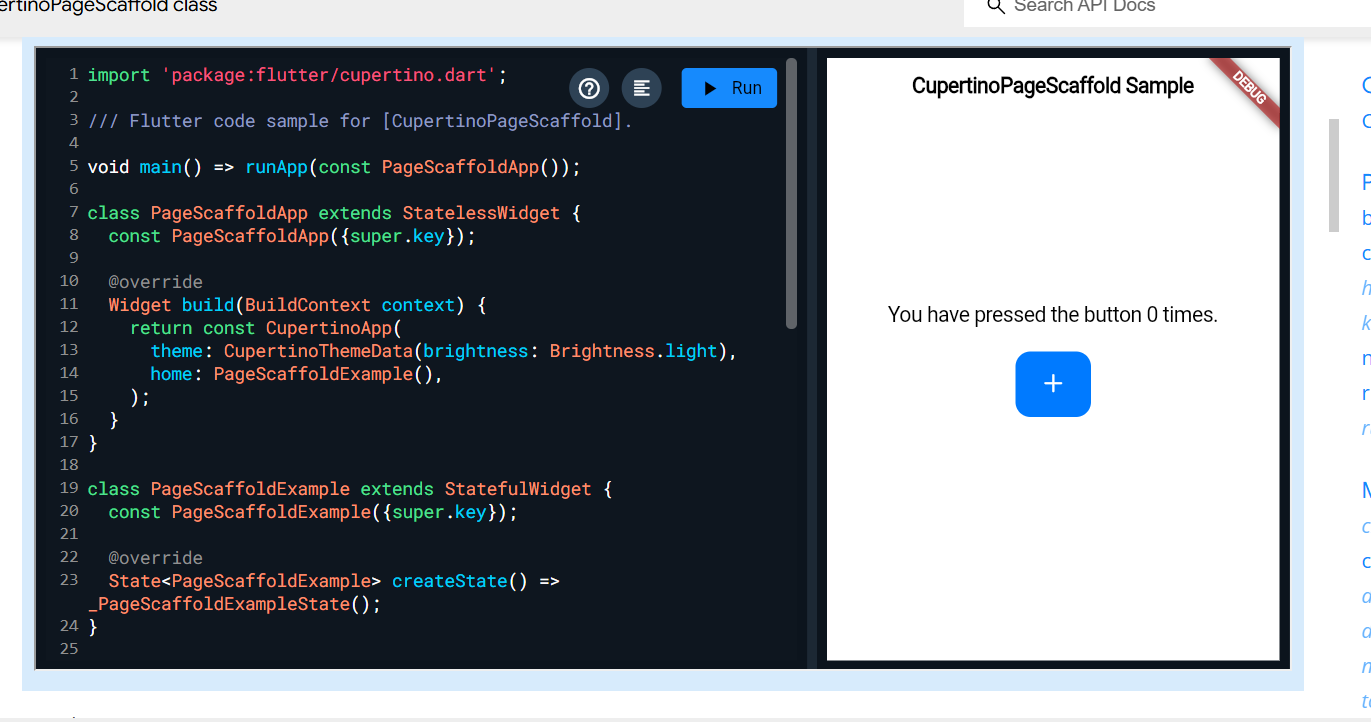
),

child: Center(

child: Text('Cupertino Page Scaffold'),

),

);



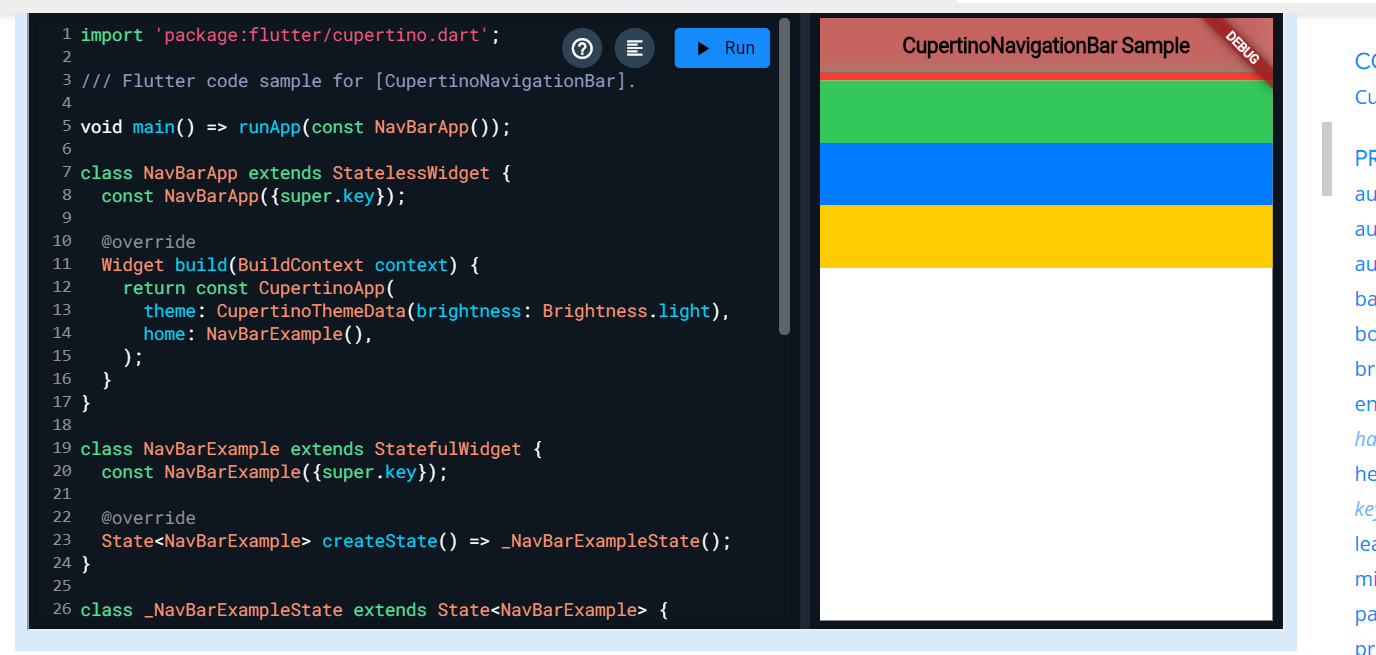
#### CupertinoNavigationBar

Навигационная панель в стиле iOS.  
**Пример:**

CupertinoNavigationBar(

middle: Text('Navigation Bar'),

);



### ****26. Flutter. Виджеты в стиле Купертино****

#### CupertinoDialog

Диалоговое окно в стиле iOS.  
**Пример:**

showCupertinoDialog(

context: context,

builder: (context) => CupertinoAlertDialog(

title: Text('Alert'),

content: Text('This is a CupertinoDialog'),

actions: [

CupertinoDialogAction(

onPressed: () => Navigator.pop(context),

child: Text('Cancel'),

),

],

),

);

#### CupertinoButton

Кнопка в стиле iOS.  
**Пример:**

CupertinoButton(

onPressed: () {},

child: Text('Press Me'),

);

#### CupertinoContextMenu

Контекстное меню, которое появляется при длительном нажатии.  
**Пример:**

CupertinoContextMenu(

actions: [

CupertinoContextMenuAction(

child: Text('Action 1'),

onPressed: () {},

),

],

child: Container(color: Colors.blue, height: 100, width: 100),

);

#### CupertinoDatePicker

Компонент для выбора даты.  
**Пример:**

CupertinoDatePicker(

mode: CupertinoDatePickerMode.date,

onDateTimeChanged: (date) {

// Обработка выбора даты

},

);

#### CupertinoTimerPicker

Компонент для выбора времени.  
**Пример:**

CupertinoTimerPicker(

mode: CupertinoTimerPickerMode.hm,

onTimerDurationChanged: (duration) {

// Обработка изменения времени

},

);

### ****27. Flutter. Виджеты в стиле Купертино****

#### CupertinoTabScaffold

Макет для организации вкладок в стиле iOS.  
**Пример:**

CupertinoTabScaffold(

tabBar: CupertinoTabBar(

items: [

BottomNavigationBarItem(icon: Icon(CupertinoIcons.home)),

BottomNavigationBarItem(icon: Icon(CupertinoIcons.settings)),

],

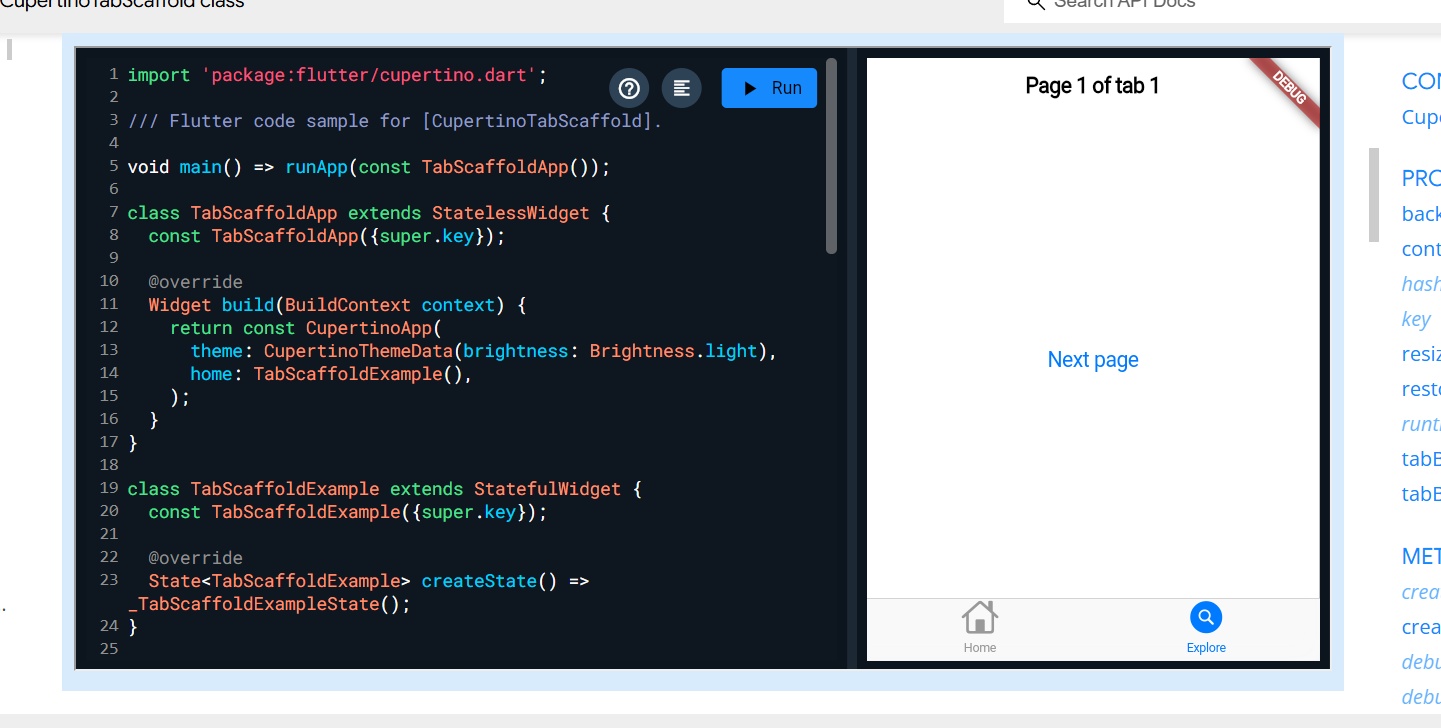
),

tabBuilder: (context, index) {

return Center(child: Text('Tab $index'));

},

);



#### CupertinoTabView

Создание отдельных экранов для вкладок. Используется внутри CupertinoTabScaffold.

#### CupertinoTextField

Поле ввода в стиле iOS.  
**Пример:**

CupertinoTextField(

placeholder: 'Enter text',

);

### ****28. Flutter. Навигация.**** PageView

PageView — виджет для создания прокручиваемых страниц.  
**Пример:**

PageView(

children: [

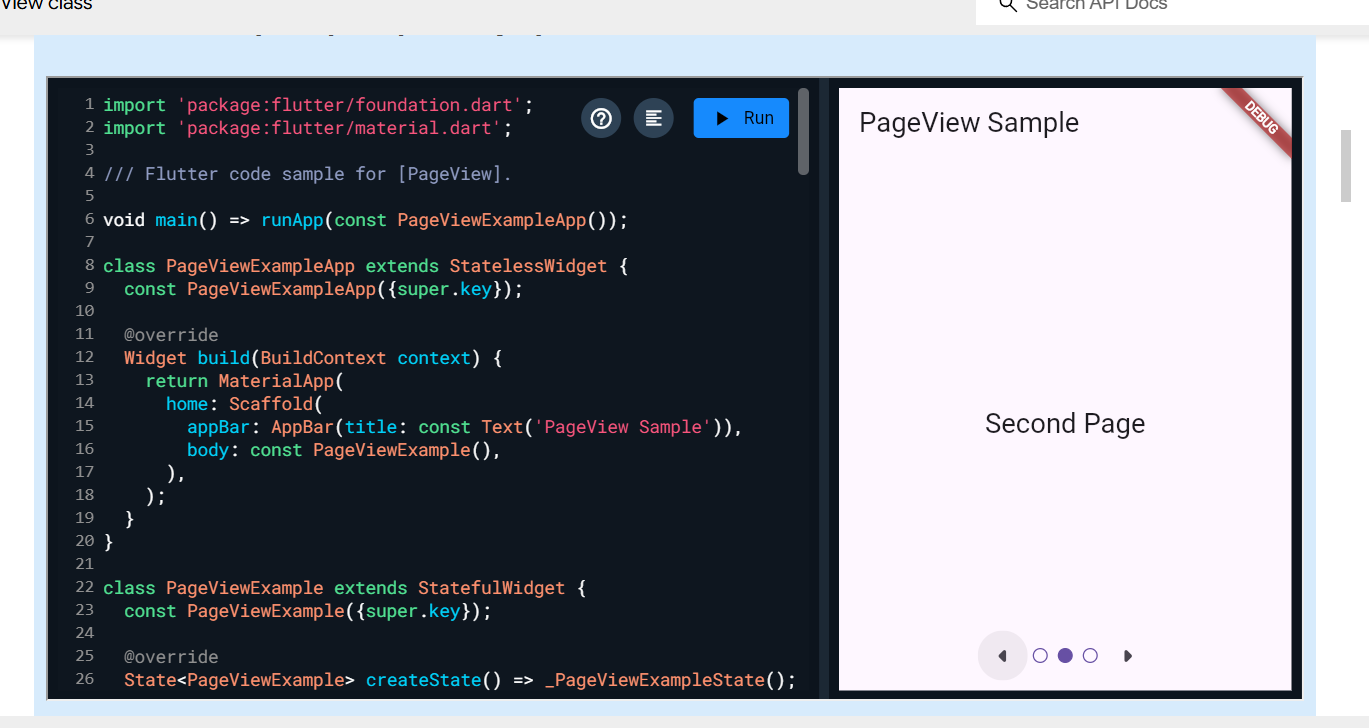
Center(child: Text('Page 1')),

Center(child: Text('Page 2')),

Center(child: Text('Page 3')),

],

);





### ****29. Flutter. Platform channel****

Platform channel — механизм для взаимодействия Flutter с нативным кодом на Android или iOS.

**Процесс создания:**

1. Создайте канал связи (MethodChannel) в коде Flutter.
2. Реализуйте обработчик вызовов в нативном коде.
3. Обеспечьте обратную связь через канал.

**Пример:**

MethodChannel channel = MethodChannel('com.example.channel');

Future<void> callNativeCode() async {

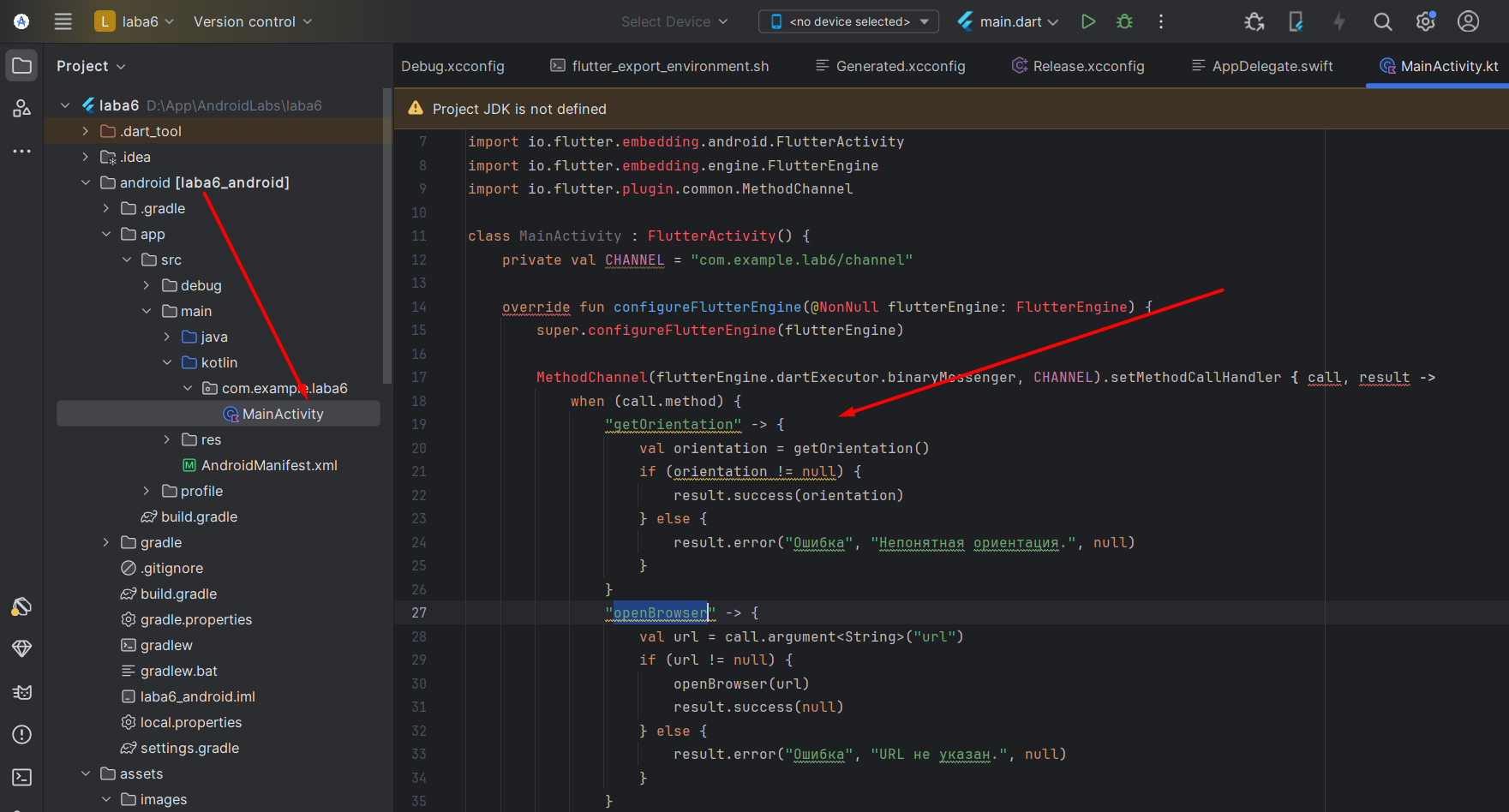
final String result = await channel.invokeMethod('nativeFunction');

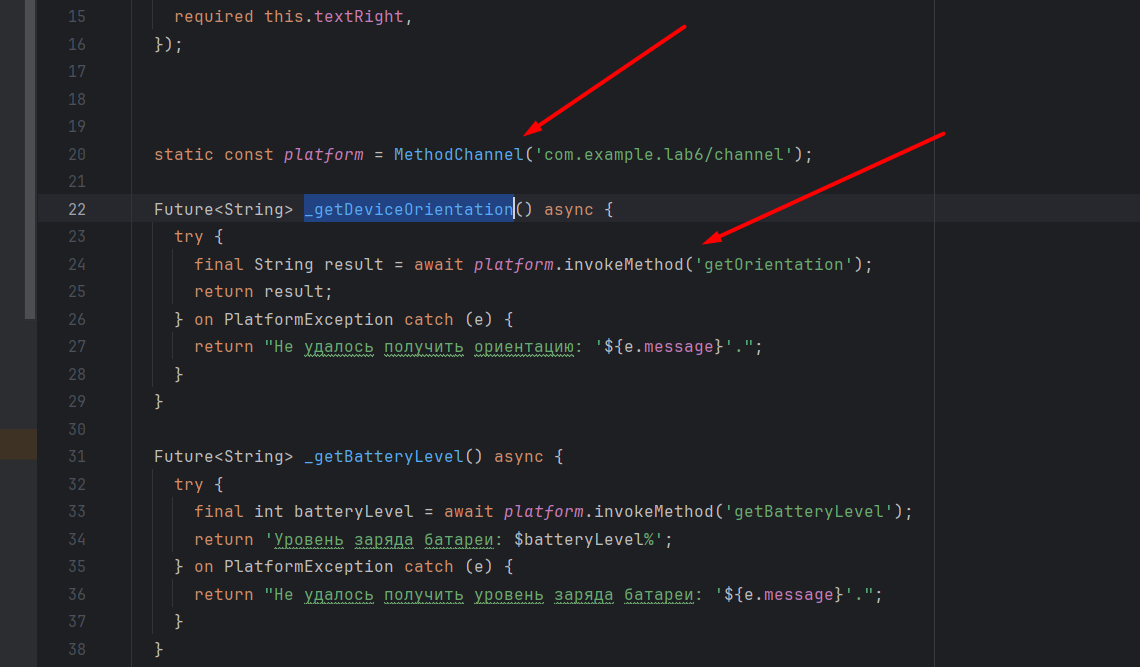
}

**Применение:**

* Используется для доступа к нативным API, которые не поддерживаются Flutter напрямую (например, Bluetooth, камера).

Как было в лабе





### ****30. Flutter. File System****

Для работы с файловой системой используется пакет dart:io или библиотека path\_provider.

**Пример чтения файла:**

import 'dart:io';

import 'package:path\_provider/path\_provider.dart';

Future<String> readFile() async {

final directory = await getApplicationDocumentsDirectory();

final file = File('${directory.path}/example.txt');

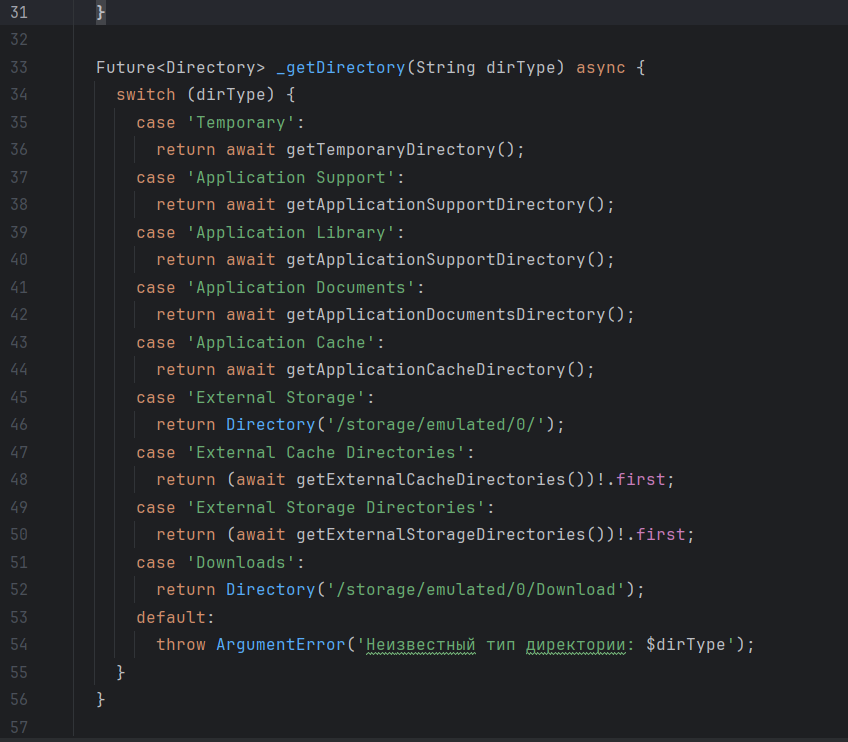
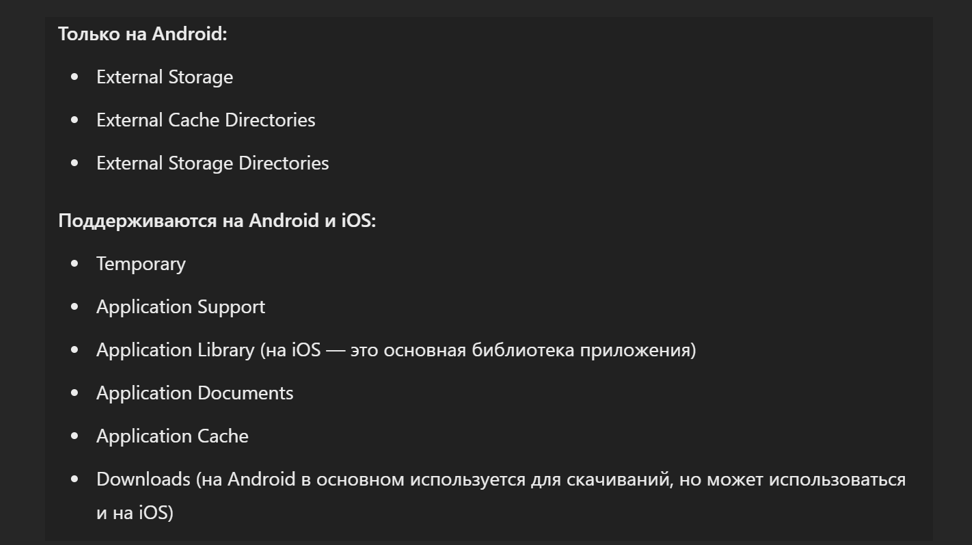
return await file.readAsString();

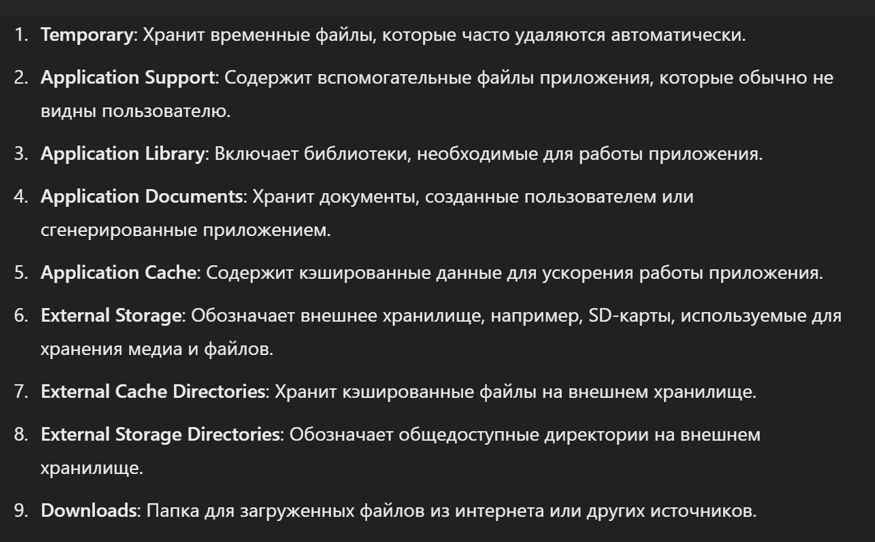
}

**Применение:**

* Хранение данных в локальных файлах.
* Чтение и запись текстовых и бинарных данных.

Какие были директории в лабе?



### ****31. Flutter. Sqflite. Управление базой данных. SharedPreferences****

#### Sqflite

Пакет для работы с локальной базой данных SQLite.

**Основные действия:**

* Создание базы данных и таблиц.
* Вставка, чтение, обновление и удаление данных (CRUD).

**Пример работы с Sqflite:**

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

Future<void> main() async {

final database = openDatabase(

join(await getDatabasesPath(), 'example.db'),

onCreate: (db, version) {

return db.execute('CREATE TABLE items(id INTEGER PRIMARY KEY, name TEXT)');

},

version: 1,

);

Future<void> insertItem(Map<String, dynamic> item) async {

final db = await database;

await db.insert('items', item);

}

Future<List<Map<String, dynamic>>> getItems() async {

final db = await database;

return await db.query('items');

}

}

#### SharedPreferences

Легковесный механизм хранения данных в формате ключ-значение.

**Пример использования:**

import 'package:shared\_preferences/shared\_preferences.dart';

Future<void> saveData(String key, String value) async {

final prefs = await SharedPreferences.getInstance();

await prefs.setString(key, value);

}

Future<String?> getData(String key) async {

final prefs = await SharedPreferences.getInstance();

return prefs.getString(key);

}

### ****32. Flutter. Permissions. Notifications****

#### Permissions

Используются для запроса доступа к камере, геолокации и другим ресурсам. Рекомендуется использовать пакет [permission\_handler](https://pub.dev/packages/permission_handler).

**Пример запроса разрешения:**

import 'package:permission\_handler/permission\_handler.dart';

Future<void> requestPermission() async {

if (await Permission.camera.request().isGranted) {

print('Permission granted');

} else {

print('Permission denied');

}

}

#### Notifications

Используются для отображения локальных или пуш-уведомлений. Пакет [flutter\_local\_notifications](https://pub.dev/packages/flutter_local_notifications) — популярное решение.

**Пример отображения локального уведомления:**

import 'package:flutter\_local\_notifications/flutter\_local\_notifications.dart';

final notifications = FlutterLocalNotificationsPlugin();

Future<void> initNotifications() async {

const androidSettings = AndroidInitializationSettings('@mipmap/ic\_launcher');

const initSettings = InitializationSettings(android: androidSettings);

await notifications.initialize(initSettings);

}

Future<void> showNotification() async {

const details = NotificationDetails(

android: AndroidNotificationDetails('channel\_id', 'channel\_name'),

);

await notifications.show(0, 'Title', 'Body', details);

}

### ****33. Flutter. Firebase. Сервисы Firebase****

**Основные сервисы Firebase:**

* **Authentication**: Аутентификация пользователей.
* **Firestore**: Облачная база данных в реальном времени.
* **Firebase Storage**: Хранение файлов.
* **Cloud Messaging (FCM)**: Пуш-уведомления.
* **Crashlytics**: Отслеживание ошибок.

**Пример использования Authentication:**

import 'package:firebase\_auth/firebase\_auth.dart';

Future<void> signIn(String email, String password) async {

try {

final userCredential = await FirebaseAuth.instance.signInWithEmailAndPassword(

email: email,

password: password,

);

print('Signed in as ${userCredential.user?.email}');

} catch (e) {

print('Error: $e');

}

}

### ****34. Flutter. FutureBuilder. StreamBuilder****

#### FutureBuilder

Обрабатывает данные из Future и обновляет интерфейс при завершении.  
**Пример:**

Future<String> fetchData() async {

await Future.delayed(Duration(seconds: 2));

return 'Data Loaded';

}

FutureBuilder<String>(

future: fetchData(),

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.waiting) {

return CircularProgressIndicator();

} else if (snapshot.hasError) {

return Text('Error: ${snapshot.error}');

} else {

return Text('Result: ${snapshot.data}');

}

},

);

#### StreamBuilder

Работает с потоками данных (Stream).  
**Пример:**

Stream<int> counterStream() async\* {

for (int i = 0; i < 5; i++) {

await Future.delayed(Duration(seconds: 1));

yield i;

}

}

StreamBuilder<int>(

stream: counterStream(),

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.active) {

return Text('Count: ${snapshot.data}');

} else {

return CircularProgressIndicator();

}

},

);

### ****35. BLoC. Преимущества и недостатки. Схема****

**BLoC** (Business Logic Component) — паттерн управления состоянием.

**Преимущества:**

* Легко тестируется.
* Четкое разделение бизнес-логики и интерфейса.
* Повторное использование кода.

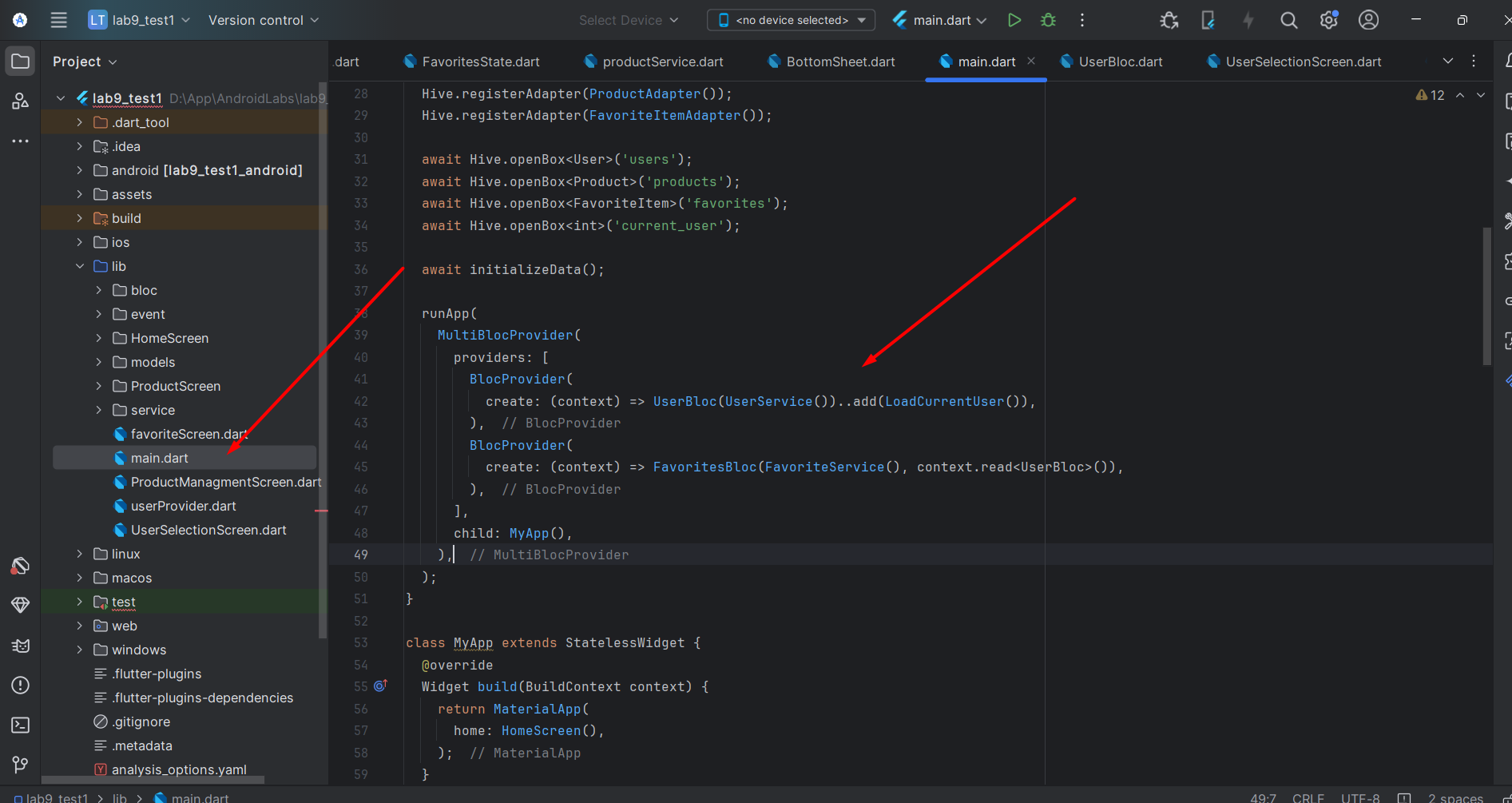
**Недостатки:**

* Более сложная структура.
* Избыточность для простых приложений.

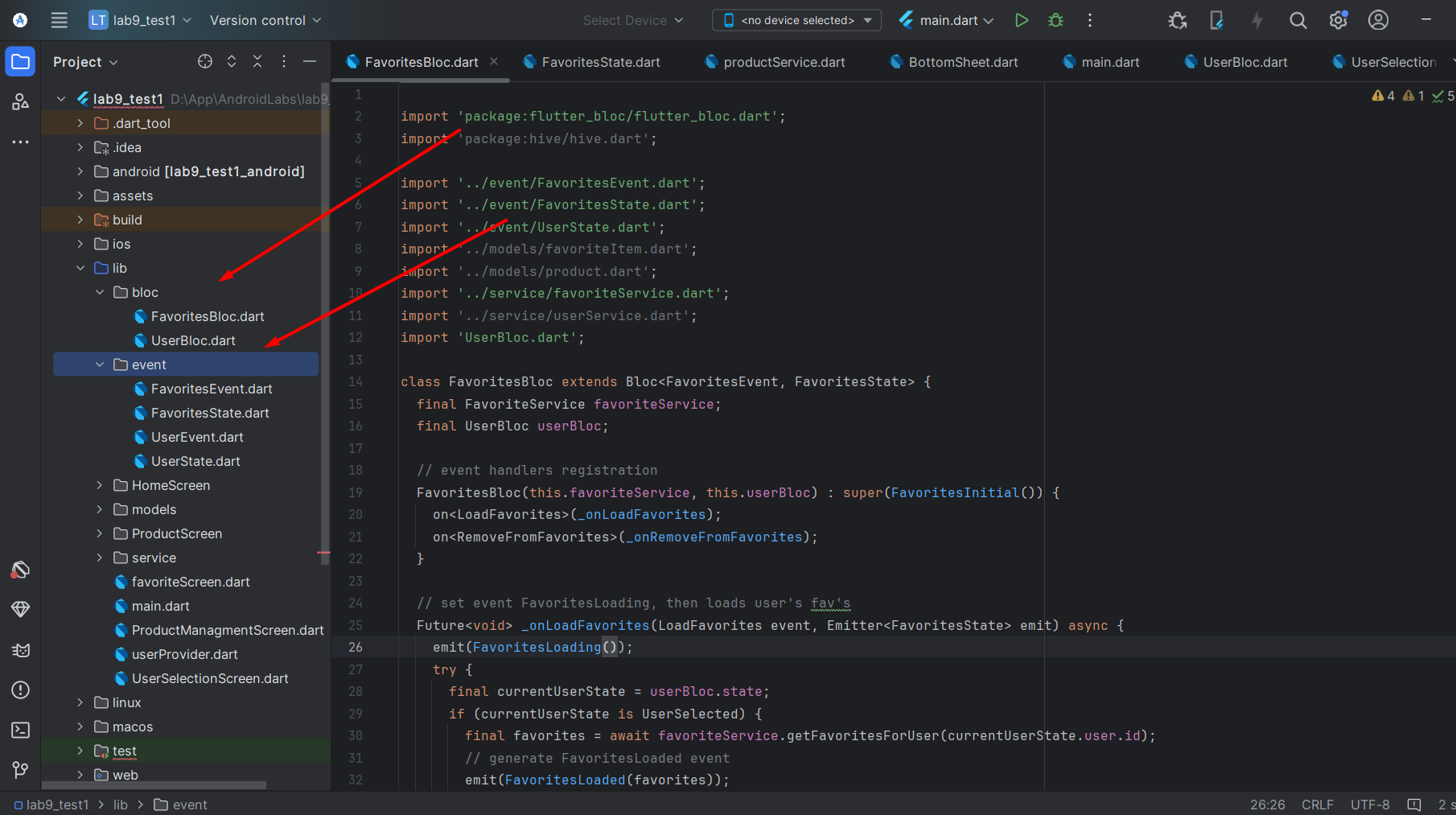
**Схема:**

1. UI отправляет события.
2. BLoC обрабатывает события и эмитирует состояния.
3. UI слушает изменения состояния.

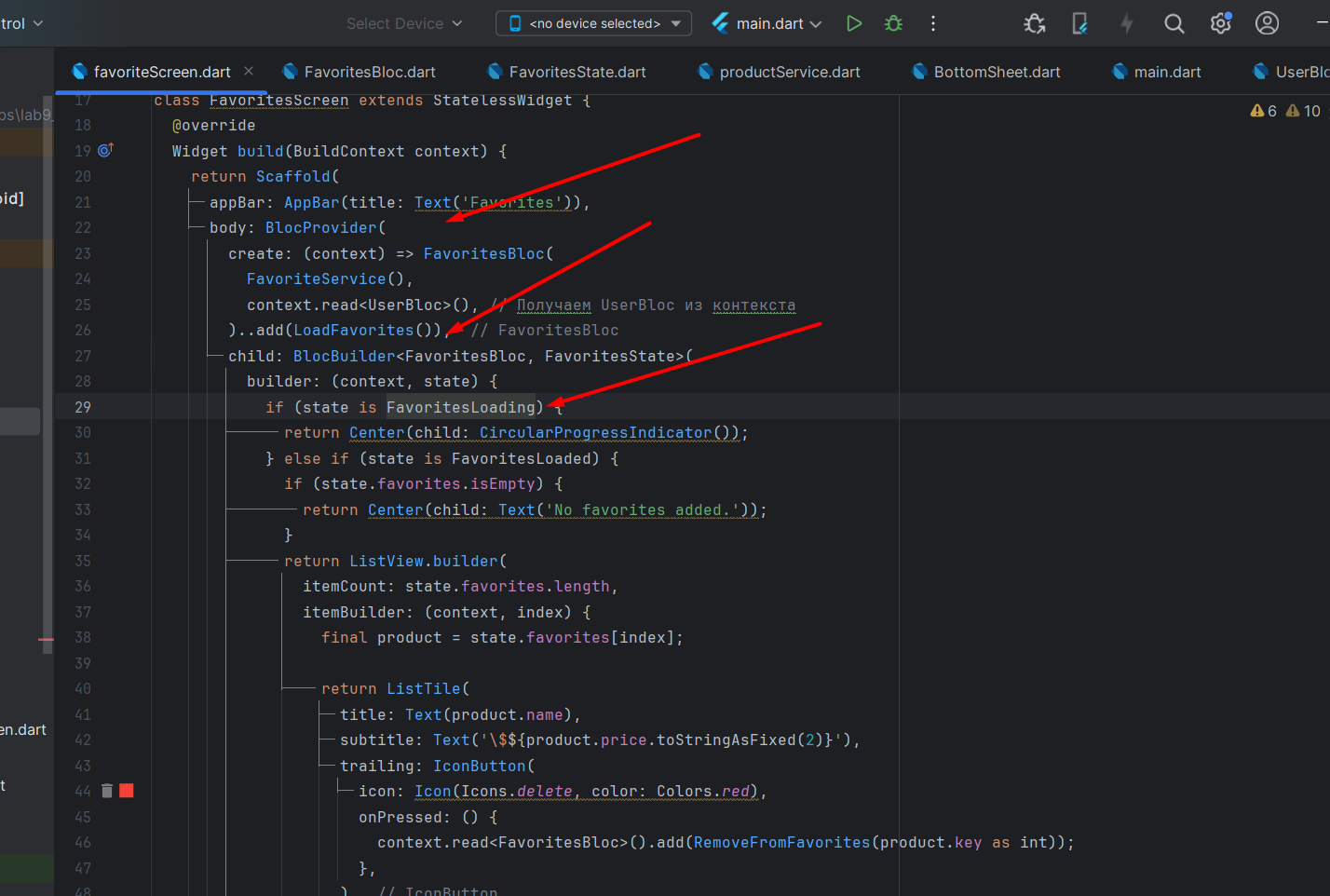
Подключаем в main

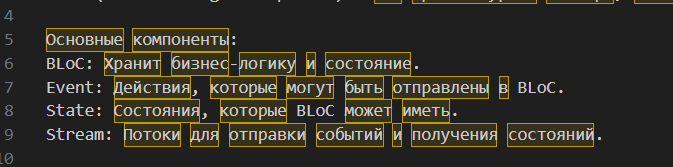


Разделение идет на блок – основаня бизнес логика, и ивенты

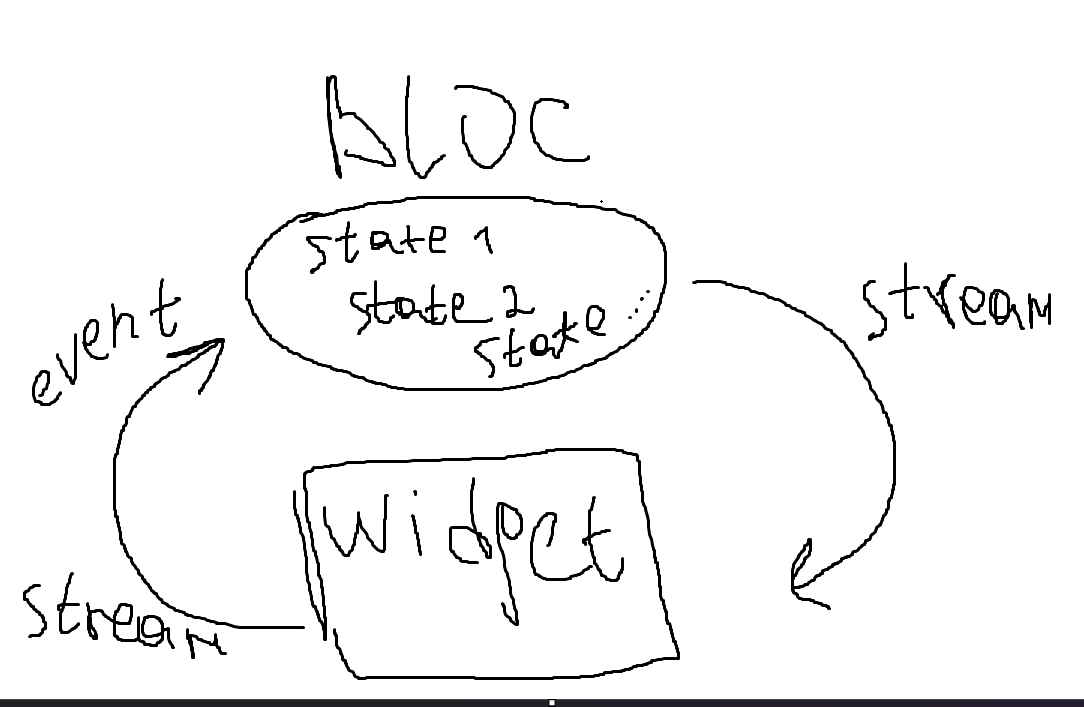


Получаем значения из блока с помощью BlocProvider.of (final yourBloc = BlocProvider.of<YourBloc>(context);) или BlocProvider context





Так понимаю подобная картина будет



### ****36. Provider. Примеры использования****

**Provider** — популярный инструмент управления состоянием.

**Пример:**

class Counter extends ChangeNotifier {

int value = 0;

void increment() {

value++;

notifyListeners();

}

}

ChangeNotifierProvider(

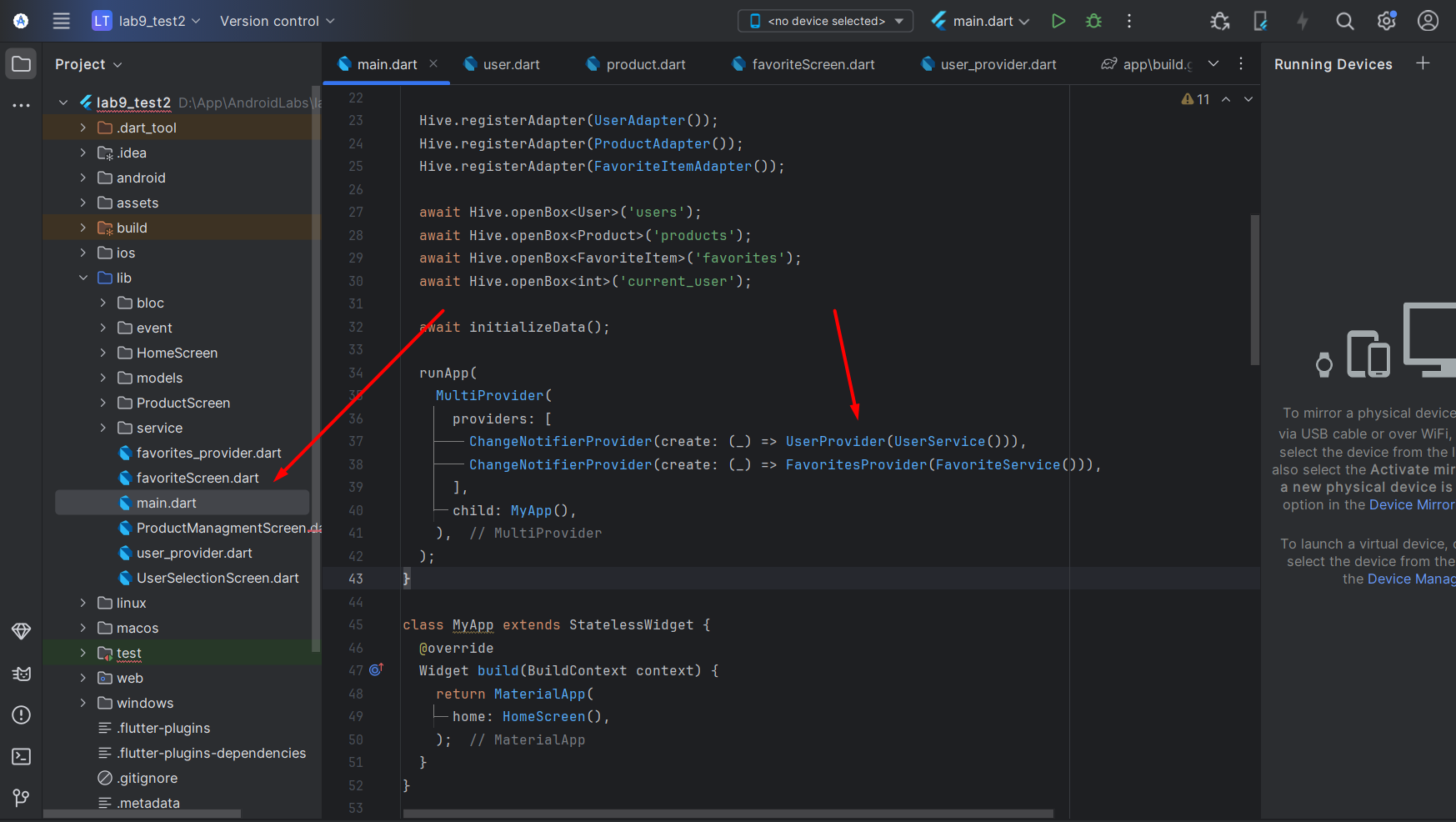
create: (\_) => Counter(),

child: Consumer<Counter>(

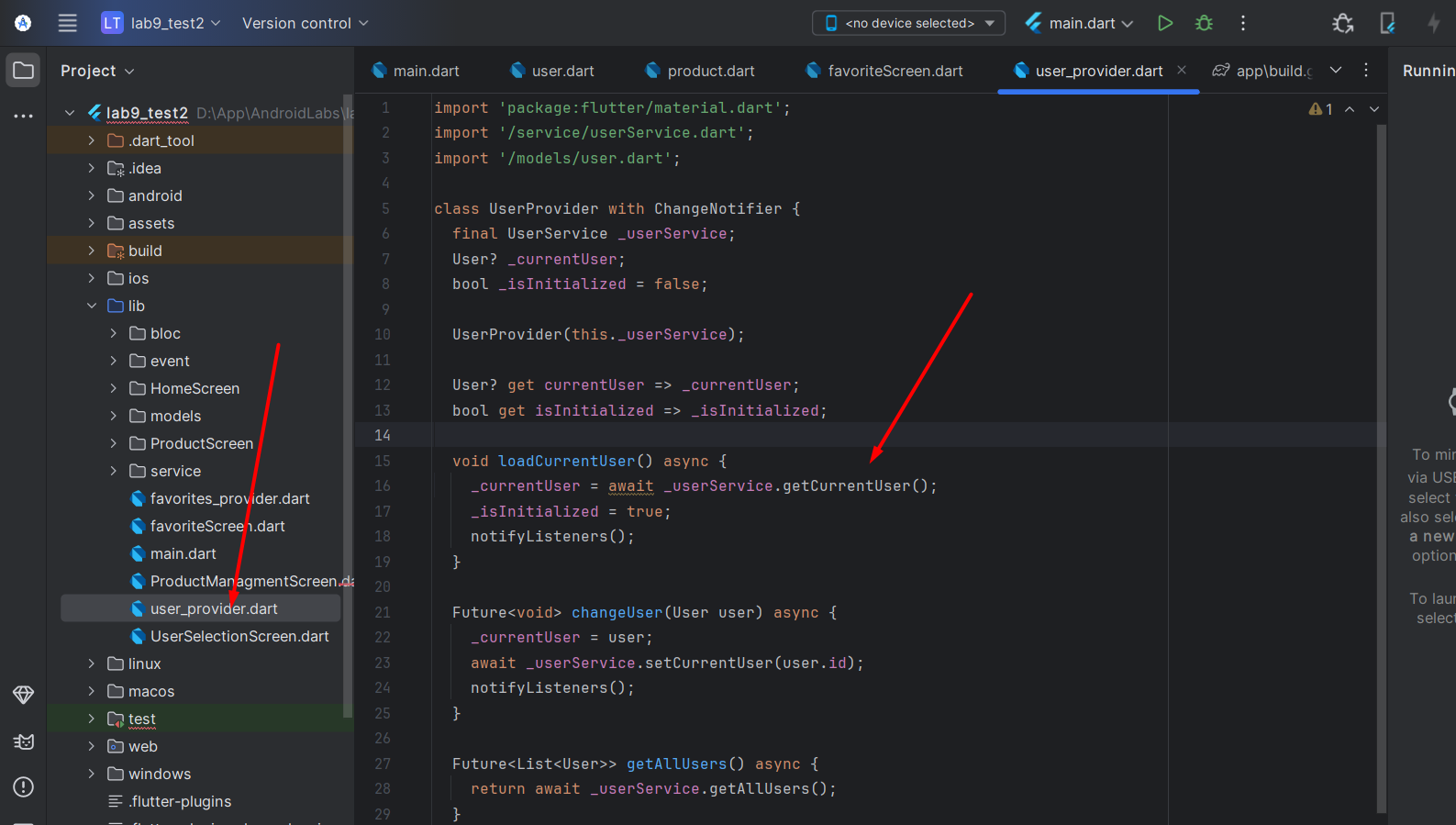
builder: (\_, counter, \_\_) => Text('Value: ${counter.value}'),

),

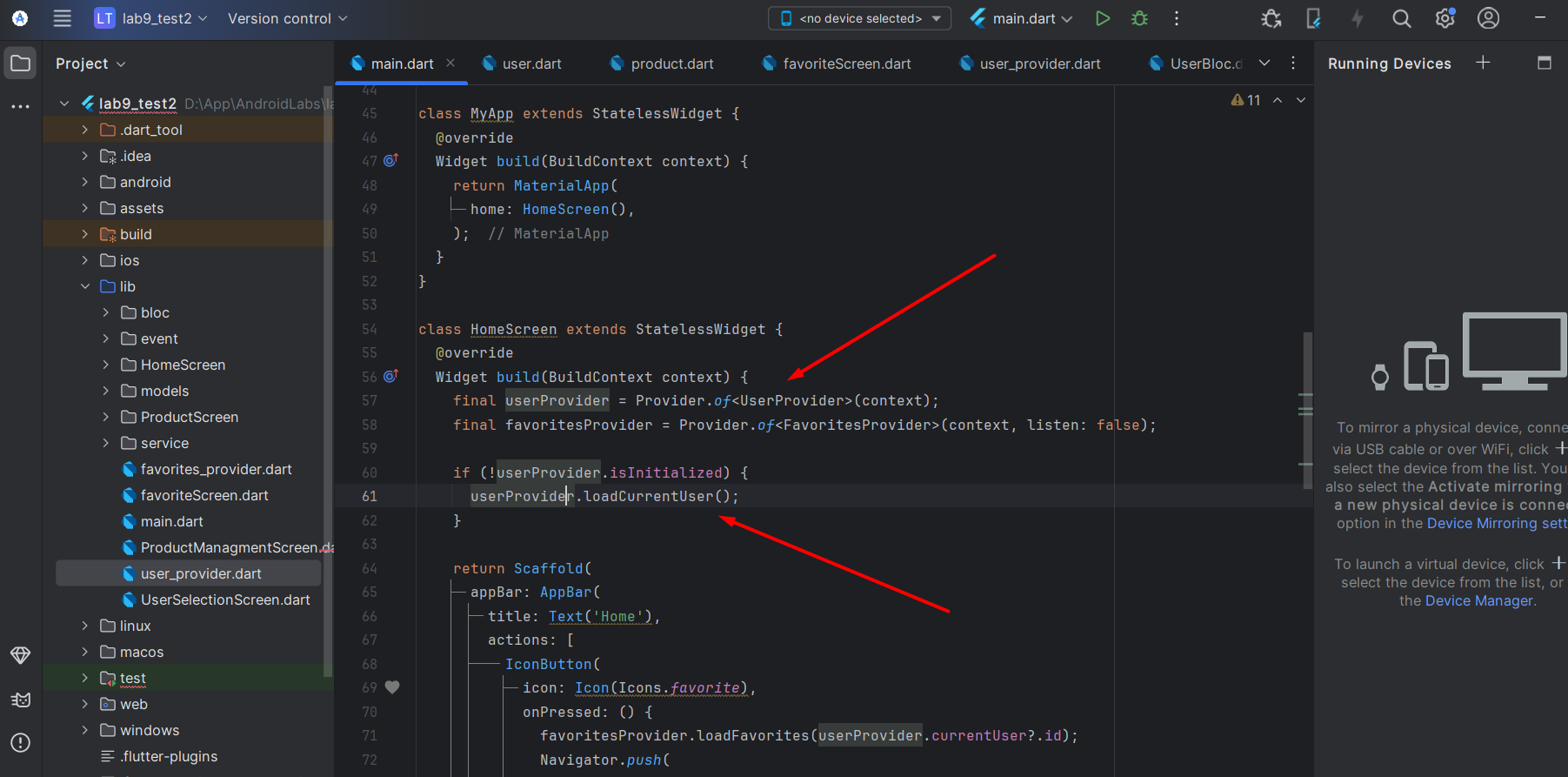
);



В main подключаем провайдер



Сам провайдер создаем определяем методы



Вызываем через provider.of

### ****37. ORM Hive. Преимущества и недостатки. Способ хранения данных****

**Hive** — легковесный NoSQL-хранилище.

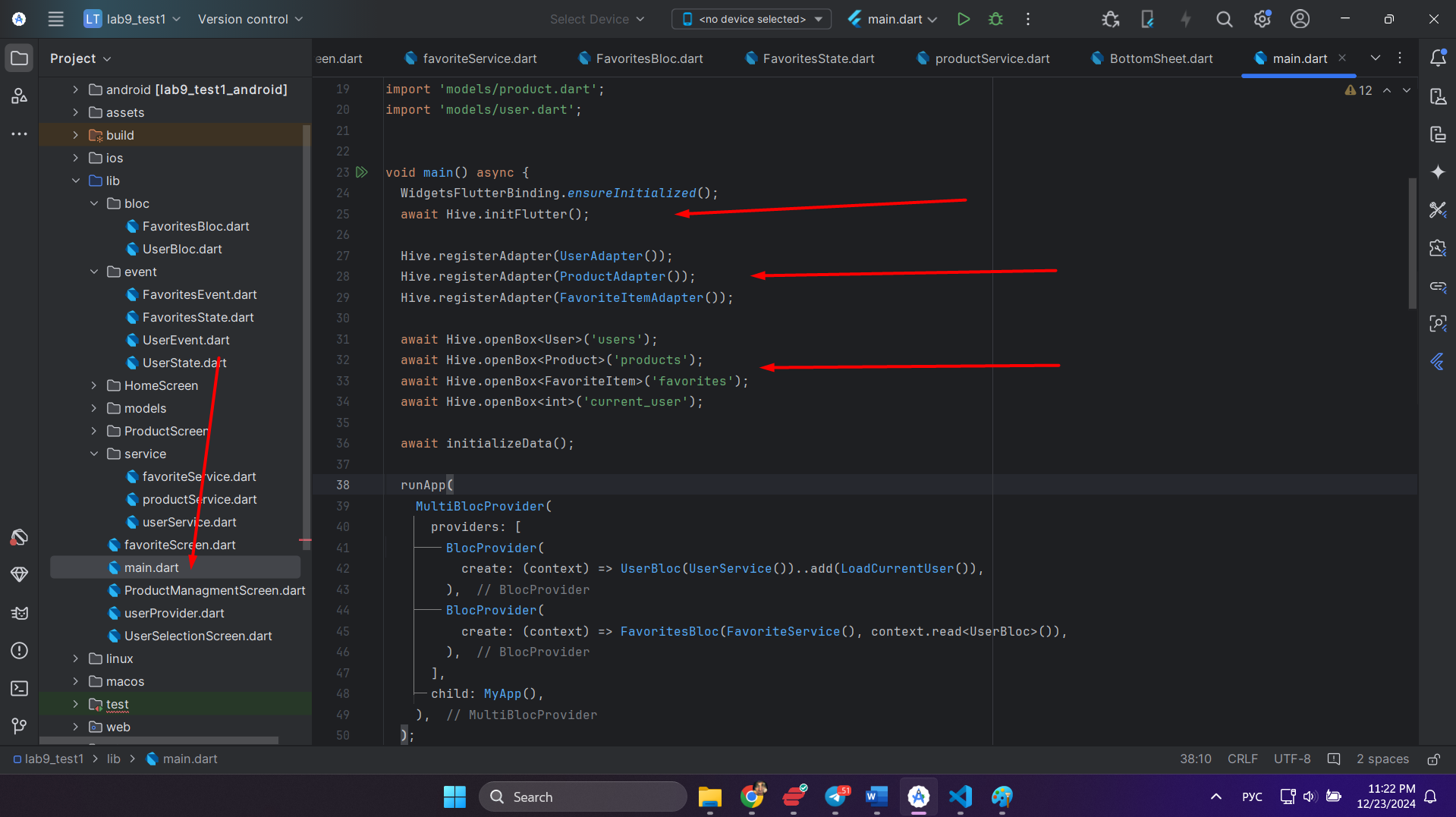
**Преимущества:**

* Высокая производительность.
* Нет необходимости в подключении нативного кода.

**Недостатки:**

* Не поддерживает сложные запросы.

**Способ хранения данных:**  
Hive сохраняет данные в бинарном формате в файлах.



Инициализиируется hive, создаются адаптеры и открываются далее можно уже добавлять или что вам надо



### ****38. Flutter Unit Tests. Mockito. Использование Mockito****

**Mockito** — библиотека для создания моков (заглушек).

**Пример теста с Mockito:**

class MockService extends Mock implements MyService {}

void main() {

final mockService = MockService();

when(mockService.getData()).thenReturn('Mock Data');

test('Test with mock data', () {

expect(mockService.getData(), 'Mock Data');

});

}

### ****39. Flutter Widget Tests. Поиск Widget’ов****

Widget-тесты проверяют поведение интерфейса.

**Пример поиска виджетов:**

testWidgets('Find a widget', (WidgetTester tester) async {

await tester.pumpWidget(MyApp());

expect(find.text('Hello'), findsOneWidget);

expect(find.byType(FlatButton), findsNothing);

});

### ****40. Flutter Integration Tests. Запуск интеграционных тестов****

Интеграционные тесты проверяют работу всего приложения. Используйте [integration\_test](https://pub.dev/packages/integration_test).

**Пример:**

import 'package:integration\_test/integration\_test.dart';

import 'package:flutter\_test/flutter\_test.dart';

void main() {

IntegrationTestWidgetsFlutterBinding.ensureInitialized();

testWidgets('Full app test', (WidgetTester tester) async {

await tester.pumpWidget(MyApp());

expect(find.text('Welcome'), findsOneWidget);

});

}