**Оглавление**

[**1)Кроссплатформенная разработка мобильных приложений. Определение, преимущества и недостатки. 3**](#_Toc159021130)

[**2)Характеристика языка программирования Dart. Преимущества и недостатки. 4**](#_Toc159021131)

[**3)Основные типы данных в языке программирования Dart. Определение и назначение типов данных: Number, String, Boolean, List, Set, Map, Rune and Grapheme Cluster, Symbol. 5**](#_Toc159021132)

[**4)Структура класса в языке программирования Dart. Типы наследования. Определение, назначение и пример использования Dart Interface. 6**](#_Toc159021133)

[**5)Определение, назначение и пример использования Dart Mixin. 7**](#_Toc159021134)

[**6)Функции в языке программирования Dart. Типы параметров и функций. 8**](#_Toc159021135)

[**7)Определение, назначение и пример использования Dart Future. 9**](#_Toc159021136)

[**8)Определение, назначение и пример использования Dart Stream. 10**](#_Toc159021137)

[**9)Определение Dart Isolate. Алгоритм создания и взаимодействия между несколькими изолятами. 11**](#_Toc159021138)

[**10)Характеристика фреймворка Flutter. Архитектура фреймворка. 12**](#_Toc159021139)

[**11)Описаний организации структуры проекта на Flutter. 13**](#_Toc159021140)

[**12)Понятия, назначение и пример использования Flutter Widget и Flutter State. 14**](#_Toc159021141)

[**13)Понятия StatelessWidget и StatefulWidget в Flutter. Отличия и примеры использования. 15**](#_Toc159021142)

[**14)Понятия Widget Tree, Element Tree и Render Tree в Flutter. Отличия и назначение. 16**](#_Toc159021143)

[**15)Определение и особенности Single-child layout Widgets в Flutter. Виджеты: Align, AspectRatio, Baseline, Center, ConstrainedBox, Container, MediaQuery. 17**](#_Toc159021144)

[**16)Определение и особенности Single-child layout Widgets в Flutter. Виджеты: Expanded, Padding, SizedBox, Transform, OverflowBox, FittedBox. 18**](#_Toc159021145)

[**17)Определение и особенности Multi-child layout Widgets в Flutter. Виджеты: Column, GridView, Stack, IndexedStack. 19**](#_Toc159021146)

[**18)Определение и особенности Multi-child layout Widgets в Flutter. Виджеты: Row, ListView, ListTile, Table, Wrap. 20**](#_Toc159021147)

[**19)Определение и особенности Component Widgets в Flutter. Виджеты: AppBar, RaisedButton, Scaffold, FloatingActionButton. 21**](#_Toc159021148)

[**21)Определение и особенности Component Widgets в Flutter. Виджеты: AlertDialog, BottomSheet, ExpansionPanel, SimpleDialog, SnackBar. 23**](#_Toc159021149)

[**22)Определение и особенности Component Widgets в Flutter. Виджеты: Card, Divider, Stepper, Chip. 24**](#_Toc159021150)

[**23)Взаимодействие виджетов в Flutter с пользователем. Виджеты: Dismissible, DragTarget, Draggable, GestureDetector, LongPressDraggable. 25**](#_Toc159021151)

[**24)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoActionSheet, CupertinoSegmentedControl, CupertinoPicker. 26**](#_Toc159021152)

[**25)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoPageTransition, CupertinoPageScaffold, CupertinoNavigationBar. 27**](#_Toc159021153)

[**26)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoDialog, CupertinoButton, CupertinoContextMenu, CupertinoDatePicker, CupertinoTimerPicker. 28**](#_Toc159021154)

[**27)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoTabScaffold, CupertinoTabView, CupertinoTextField, CupertinoButton. 29**](#_Toc159021155)

[**28)Навигация в приложении Flutter. Определение и назначение Navigator, MaterialPageRoute, CupertinoPageRoute. 30**](#_Toc159021156)

[**29)Понятие Flutter Platform Channel. Процесс создания и способы применения. 31**](#_Toc159021157)

[**30)Алгоритм работы с файловой системой в Flutter. 32**](#_Toc159021158)

[**31)Назначение библиотек Sqflite и SharedPreference в Flutter. 33**](#_Toc159021159)

[**32)Принцип обработки разрешений в приложениях Flutter. Инструменты для управления разрешениями. 34**](#_Toc159021160)

[**33)Механизм отправки и обработки уведомлений в Flutter. Способы и инструменты. 35**](#_Toc159021161)

[**34)Понятие Firebase. Виды и назначение сервисов Firebase, доступные для интеграции с Flutter. 36**](#_Toc159021162)

[**35)Понятия FutureBuilder и StreamBuilder в Flutter. Принципы работы и примеры использования. 37**](#_Toc159021163)

[**36)Структура и назначение архитектурного шаблона Bloc в Flutter. 38**](#_Toc159021164)

[**37)Преимущества использования пакета Provider в Flutter. Применение Provider для управления состоянием. 39**](#_Toc159021165)

[**38)Принцип использования библиотеки Hive для локального хранения данных в приложениях Flutter. 40**](#_Toc159021166)

[**39)Определение и назначение юнит-тестов в Flutter. Роль библиотеки Mockito в тестировании. 41**](#_Toc159021167)

[**40)Тестирование виджетов в Flutter. Методы поиска и проверки конкретных виджетов в приложении. 42**](#_Toc159021168)

[**41)Интеграционное тестирование в приложениях Flutter. Инструменты для проверки взаимодействия компонентов. 43**](#_Toc159021169)

1)Кроссплатформенная разработка мобильных приложений. Определение, преимущества и недостатки.

Кроссплатформенная разработка мобильных приложений - это процесс создания приложений, которые могут работать на разных операционных системах, таких как iOS и Android, с использованием единого кодовой базы. Вместо того, чтобы разрабатывать отдельные приложения для каждой платформы, разработчики могут использовать кроссплатформенные фреймворки, инструменты и языки программирования, чтобы создавать приложения, которые могут быть запущены на разных платформах.

Преимущества кроссплатформенной разработки мобильных приложений:

1.Повышенная эффективность: Разработка единой кодовой базы для разных платформ позволяет сократить время и затраты на разработку. Разработчики могут использовать один и тот же код для создания приложений для разных платформ, что уменьшает необходимость в разработке отдельных версий для каждой ОС.

2.Ускоренная разработка: Кроссплатформенные фреймворки предлагают инструменты и функциональность, которые упрощают разработку мобильных приложений. Они предоставляют готовые компоненты интерфейса пользователя, доступ к аппаратным возможностям устройств и другие функции, что позволяет разработчикам быстрее создавать приложения.

3.Единый код: Разработка с использованием кроссплатформенных фреймворков позволяет иметь единую кодовую базу, что значительно упрощает поддержку и обновление приложений. Изменения в коде применяются к приложениям на всех платформах одновременно, что упрощает управление и обновление приложений.

Недостатки кроссплатформенной разработки мобильных приложений:

1.Ограниченные возможности: Некоторые кроссплатформенные фреймворки могут иметь ограниченный доступ к функциям и возможностям операционных систем. Это может ограничить функциональность и производительность приложений, особенно если требуются специфичные функции, доступные только на конкретной платформе.

2.Производительность: Кроссплатформенные фреймворки обычно имеют некоторую накладную нагрузку на производительность по сравнению с нативной разработкой. Это может быть особенно заметно в случае сложных и ресурсоемких приложений, таких как игры или графически интенсивные приложения.

Зависимость от фреймворка: При использовании кроссплатформенных фреймворков разработчики ограничены функциями и инструментами, предоставляемыми выбранным фреймворком. Это может создать зависимость от фреймворка и ограничить возможности разработчиков при необходимости реализации специфичных функций или оптимизации приложения.

2)Характеристика языка программирования Dart. Преимущества и недостатки.

Dart - это язык программирования, разработанный компанией Google, который широко используется для создания мобильных, веб- и серверных приложений. Вот некоторые характеристики Dart, а также его преимущества и недостатки:

**Характеристики Dart:**

1.Современный и объектно-ориентированный **(**поддерживает основные принципы объектно-ориентированного программирования, такие как наследование, полиморфизм и инкапсуляция.)

2.Компилируемый:компилируется в нативный код для достижения высокой производительности.

3.Кроссплатформенность.

**Преимущества Dart:**

1.Простой и понятный синтаксис

2.Высокая производительность: Благодаря компиляции в нативный код, обеспечивает высокую производительность при выполнении приложений. Также предлагает эффективное управление памятью и оптимизации

3.Фреймворк Flutter: Dart является основным языком программирования для фреймворка Flutter, который предоставляет набор инструментов для разработки кроссплатформенных мобильных и веб-приложений. Flutter имеет мощные возможности для создания красивого пользовательского интерфейса и обеспечивает быструю разработку приложений.

**Недостатки Dart:**

1.Ограниченная экосистема: Dart имеет более ограниченную экосистему и меньшее количество библиотек и инструментов, по сравнению с языками, такими как JavaScript или Python.

2.Ограничения на выбор платформы: поддерживает только iOS, Android и веб.

3.Относительная новизна: не имеет такого широкого сообщества разработчиков и ресурсов для обучения и поддержки. Это может ограничить доступность некоторых ресурсов и инструментов, которые могут быть доступны для более популярных языков.

4.Отсутствие нативной поддержки: В некоторых случаях может потребоваться использование дополнительных инструментов или библиотек для доступа к нативным функциям и API платформы, таким как доступ к определенным функциям устройства или интеграция с ОС-специфичными возможностями. Это может потребовать дополнительной работы и усилий со стороны разработчика.

3)Основные типы данных в языке программирования Dart. Определение и назначение типов данных: Number, String, Boolean, List, Set, Map, Rune and Grapheme Cluster, Symbol.

1.List (список): Тип данных List представляет упорядоченную коллекцию элементов. Список может содержать элементы разных типов данных, и его размер может быть изменяемым. Элементы списка нумеруются с 0. Например, List<int> numbers = [1, 2, 3]; создаст список чисел.

2.Set (множество): Тип данных Set представляет неупорядоченную коллекцию уникальных элементов. Множество не позволяет дублирование элементов, и порядок элементов не гарантируется. Например, Set<String> fruits = {'яблоко', 'банан', 'апельсин'}; создаст множество фруктов.

3.Map (отображение): Тип данных Map представляет ассоциативный массив, также известный как словарь или хэш-таблица. Он состоит из пар ключ-значение, где каждый ключ является уникальным. Пары ключ-значение могут быть разных типов. Например, Map<String, int> ages = {'Алиса': 25, 'Боб': 30}; создаст отображение, где ключи - имена, а значения - возраст.

4.Rune и Grapheme Cluster (символы): Rune и Grapheme Cluster используются для работы с символами, особенно в многоязычных приложениях. Rune представляет символ Unicode, а Grapheme Cluster представляет графемный кластер, который может быть составным символом. Эти типы данных используются, когда требуется более точная работа с символами, например, для манипуляции текстовыми данными.

5.Symbol (символ): Тип данных Symbol используется для представления идентификаторов, таких как имена функций или символы операций. Символы в Dart представляются с помощью символической строки, начинающейся с символа #. Например, #myFunction или #+.

4)Структура класса в языке программирования Dart. Типы наследования. Определение, назначение и пример использования Dart Interface.

// Поля (состояние) класса

// Конструкторы класса с инициализацией полей и его код

// Методы (поведение) класса и код методов

Типы наследования:

Одиночное наследование: В Dart класс может наследовать только один другой класс. Это означает, что класс наследует все поля и методы базового класса. class ChildClass extends ParentClass

Множественное наследование интерфейсов: В Dart класс может реализовывать несколько интерфейсов. Интерфейс определяет набор методов, которые класс должен реализовать. implements Interface1, Interface2

**Dart Interface** - класс, который выступает в роли интерфейса. если класс-интерфейс определяет какие-нибудь поля и методы, то класс, реализующий данный интерфейс, должен также определить эти поля и методы. Он не содержит реализации методов, только их сигнатуры.

5)Определение, назначение и пример использования Dart Mixin.

Миксин - это класс или тип, который содержит фрагменты кода, которые могут быть включены в другие классы. Миксины позволяют добавить поведение и функциональность к классам, не нарушая ограничения одиночного наследования. Они предоставляют модульность и возможность повторного использования кода, что способствует улучшению структуры и читаемости кода.

Назначение миксинов:

1.Предоставление поведения: Миксины могут содержать методы, поля и другиx члены класса, которые могут быть добавлены к другим классам для предоставления дополнительного поведения.

Композиция кода: Миксины позволяют комбинировать фрагменты кода из разных источников, что позволяет создавать классы с различными комбинациями поведения. Это способствует повторному использованию кода и упрощает его поддержку и расширение.

Например есть два миксина Jumping и Swimming с методами jump(выводит ‘прыгаю’) и swim (выводит ‘плаваю’) и класс Animal. У нас есть объект которому соответствуют наши миксины и класс.

Создаем его: class Frog extends Animal with Jumping, Swimming {}

И в мейне при обращении к обьекту Frog мы также можем выводить методы миксинов

frog.jump(); // Выводит 'Прыгаю!'

frog.swim(); // Выводит 'Плаваю!'

6)Функции в языке программирования Dart. Типы параметров и функций.

Функции в языке программирования Dart - это блоки кода, которые выполняют определенные действия и могут принимать и возвращать значения. Функции могут быть определены с помощью ключевого слова func или с помощью стрелочного синтаксиса =>

Позиционные параметры: Значения передаются в функцию в определенном порядке и соответствуют позициям параметров в определении функции.

Именованные параметры: Параметры передаются в функцию с использованием их имен

Параметры со значениями по умолчанию: Параметры могут иметь значения по умолчанию, которые будут использоваться, если значения не будут переданы при вызове функции.

параметры могут быть объявлены как необязательные с использованием знака вопроса (?)

Функции без возвращаемого значения (void):

Функции с возвращаемым значением:

Анонимные функции (лямбда-выражения):

Функции высшего порядка (функции, которые принимают другие функции в качестве параметров или возвращают их.

7)Определение, назначение и пример использования Dart Future.

Future - это промис (promise) или обещание, что определенное значение будет доступно в будущем.

Future позволяет работать с данными, которые еще не доступны, но могут стать доступными позже.

Назначение Future - это упростить асинхронное программирование в Dart, то есть программирование, в котором несколько задач могут выполняться одновременно без блокировки основного потока исполнения. Future позволяет организовать цепочки асинхронных действий, обрабатывать ошибки, ждать завершения нескольких операций и т.д

Обработка результатов: Future позволяет задать коллбэки (обработчики) для обработки результатов асинхронной операции. Когда операция завершается, результат (или исключение) передается указанным обработчикам.

Future<int> fetchNumber() {

return Future.delayed(Duration(seconds: 2), () => 42);

}

8)Определение, назначение и пример использования Dart Stream.

Stream - это поток данных, представляющий последовательность асинхронных событий. Он может быть использован для организации и обработки асинхронных операций и событий в программе. Stream предоставляет механизм для получения данных по мере их генерации и обработки этих данных с помощью коллбэков или методов, таких как listen().

Назначение Stream:

·Получение данных по мере генерации: Stream позволяет получать данные по мере их генерации или поступления. Это позволяет обрабатывать данные по-частям или в реальном времени, не ожидая завершения асинхронной операции.

Обработка данных: Stream предоставляет различные методы и коллбэки для обработки данных, полученных из потока. Вы можете использовать метод listen() для установки обработчиков событий, которые будут вызываться при поступлении новых данных или событий.

Допустим можно создать поток, который генерирует числа от 1 до 100 с задержкой секунду и потом выводить эти числа после того, как они сгенерировались через .listen((data){print()}).

9)Определение Dart Isolate. Алгоритм создания и взаимодействия между несколькими изолятами.

Isolate - это изолированное окружение выполнения кода, которое работает независимо от других изолятов и основного потока. Каждый изолят имеет свою собственную память и исполняет код параллельно с другими изолятами. Isolate позволяет выполнять вычисления в отдельных потоках, обрабатывать длительные операции и улучшать производительность приложения.

Назначение Isolate:

·Параллельное выполнение: Isolate позволяет выполнять код параллельно с другими изолятами и основным потоком.

Изоляция и безопасность: Каждый Isolate выполняется в изолированном окружении, имеет свою собственную память и не может напрямую взаимодействовать с другими изолятами. Это обеспечивает безопасность и предотвращает возможные конфликты данных или состояний.

Алгоритм создания и взаимодействия между несколькими изолятами состоит из следующих шагов:

* Создать объект ReceivePort в главном изоляте, который будет принимать сообщения от других изолятов.
* Создать новый изолят с помощью статического метода Isolate.spawn, передав ему функцию, которая будет выполняться в новом изоляте, и sendPort объекта ReceivePort, который будет использоваться для отправки сообщений в главный изолят.
* Добавить обработчик событий к объекту ReceivePort, который будет вызываться при получении сообщений от других изолятов.
* В функции, которая выполняется в новом изоляте, создать объект SendPort, который будет получен из параметра функции, и использовать его для отправки сообщений в главный изолят.
* Повторить шаги 2-4 для каждого нового изолята, который нужно создать.
* Закрыть объект ReceivePort и уничтожить изоляты, когда они больше не нужны.

10)Характеристика фреймворка Flutter. Архитектура фреймворка.

***1.Кросс-платформенность***: обеспечивает единообразный внешний вид и поведение приложений на разных платформах.

***2.Богатый набор виджетов***

***3.Быстрая разработка:*** (Hot Reload)

***4.Высокая производительность:*** Благодаря своей архитектуре и использованию собственного движка рендеринга. Он позволяет достичь плавной анимации, быстрого отклика пользовательского интерфейса и эффективного использования ресурсов устройства.

***Доступ к нативным возможностям:*** С помощью специальных пакетов и платформенных каналов можно взаимодействовать с функциональностью операционной системы, такой как камера, геолокация, датчики и другие возможности устройства.

Архитектура:

1.Виджеты: являются основными строительными блоками пользовательского интерфейса. Отображаются и взаимодействуют с пользователем.

2.Элементы: низкоуровневые объекты, которые связывают виджеты с фактическими платформенными компонентами и управляют обновлением пользовательского интерфейса. Отслеживают изменения состояния виджетов и обновляют только изменившиеся части интерфейса, минимизируя затраты по рендерингу и обновлению.

3.Рамка: предоставляет инструменты и функциональность для разработки приложений. Рамка включает в себя набор библиотек, классов и модулей, которые облегчают разработку, управление состоянием и обработку событий.

4.Движок рендеринга: Flutter использует свой собственный движок рендеринга, называемый Skia, для отрисовки пользовательского интерфейса на разных платформах.

5. Движок Flutter: является ядром фреймворка и отвечает за выполнение и управление Flutter-приложениями.

11)Описаний организации структуры проекта на Flutter.

Папка "lib": Это основная папка проекта, где находится весь исходный код Flutter приложения.

Папки и файлы для тестирования.

Папки android, ios, linux, macos, web и windows в проекте Flutter содержат код и настройки, специфичные для каждой платформы.

Файл [pubspec.yaml](https://dart.dev/tools/pub/pubspec) относится к важным компонентам проекта Flutter или Dart. Он определяет конфигурацию проекта и используется, чтобы определить зависимости проекта, версии библиотек, настройки сборки и другие свойства проекта.

12)Понятия, назначение и пример использования Flutter Widget и Flutter State.

**1.Flutter Widget:**

Flutter Widget - это базовый строительный блок пользовательского интерфейса во Flutter. Он представляет собой объединенную и реиспользуемую часть интерфейса, которая может включать в себя другие виджеты. Виджеты могут быть контейнерами, макетами, текстовыми элементами, кнопками и многими другими элементами интерфейса.

Назначение Flutter Widget заключается в определении и отображении пользовательского интерфейса. Они описывают, как выглядит и ведет себя каждая часть приложения. Виджеты могут быть размещены в древовидной структуре, где каждый виджет имеет родительский виджет и может содержать дочерние виджеты. Это позволяет создавать сложные иерархии виджетов для построения интерфейса.

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text('My App'),

),

body: Center(

child: Text('Hello, Flutter!'),

),

),

);

**Flutter State** - это концепция, которая позволяет отслеживать и обновлять изменения данных в приложении. Он представляет собой информацию, которая может изменяться во время выполнения приложения и влиять на отображение пользовательского интерфейса.

onPressed: incrementCounter //бахнуть куда-нибудь на кнопку допустим

а он void incrementCounter() {

setState(() {

counter++;

}); //ну и counter сразу 0 пусть будет.

Text(

'Counter: $counter',

style: TextStyle(fontSize: 48),

), //ну и бахнуть блок для вывода counter.

13)Понятия StatelessWidget и StatefulWidget в Flutter. Отличия и примеры использования.

StatelessWidget - это класс виджета, который не имеет изменяемого состояния.

·Не имеет изменяемого состояния.

·Построен только на основе входных параметров (свойств).

Не требует реализации методов createState() и build().

StatefulWidget - это класс виджета, который имеет изменяемое состояние.

·Имеет изменяемое состояние.

·Реализует методы createState() и build().

Может использоваться для обновления интерфейса на основе внутренних или внешних событий.

пример использования StatefulWidget мы меняем значения counter выше и меняется значение на единицу. А в StatelessWidget он бы не изменялся

14)Понятия Widget Tree, Element Tree и Render Tree в Flutter. Отличия и назначение.

Widget Tree - это иерархическая структура, состоящая из виджетов в Flutter. Она описывает отношения и компоновку виджетов, где каждый виджет имеет родительский виджет и может иметь дочерние виджеты. Widget Tree определяет структуру пользовательского интерфейса и взаимодействия между виджетами

Element Tree - это внутреннее представление виджетов во время выполнения в Flutter.

Render Tree - это структура, которая представляет отрисовку виджетов на экране. Он содержит информацию о том, как каждый виджет должен быть отображен и какие элементы интерфейса должны быть отрисованы. Render Tree описывает, как виджеты преобразуются в конечный растровый (пиксельный) интерфейс на экране.

в целом

·Widget Tree определяет структуру и компоновку виджетов в пользовательском интерфейсе.

·Element Tree представляет внутреннее представление виджетов и используется для обновления интерфейса при изменении состояния.

Render Tree описывает отрисовку виджетов на экране и используется для обновления отображения при изменении состояния.

15)Определение и особенности Single-child layout Widgets в Flutter. Виджеты: Align, AspectRatio, Baseline, Center, ConstrainedBox, Container, MediaQuery.

Single-child layout Widgets в Flutter - это виджеты, которые принимают один дочерний виджет и управляют его размером, позицией или другими аспектами компоновки. Они полезны для создания сложных интерфейсов из простых элементов[1](https://education.yandex.ru/handbook/flutter/article/widgets-layout)[2](https://docs.flutter.dev/ui/widgets/layout). Вот некоторые из них:

* Align - виджет, который выравнивает своего ребенка внутри себя и опционально изменяет свой размер в зависимости от размера ребенка. Выравнивание определяется свойством alignment, которое принимает значение из перечисления Alignment или AlignmentDirectional[3](https://stackoverflow.com/questions/65914228/what-is-the-difference-between-the-single-child-layout-widget-multi-child-layo).
* AspectRatio - виджет, который пытается задать ребенку определенное соотношение сторон. Соотношение сторон определяется свойством aspectRatio, которое представляет собой отношение ширины к высоте.
* Baseline - виджет, который позиционирует своего ребенка относительно базовой линии текста. Базовая линия определяется свойствами baseline и baselineType, которые указывают расстояние от верхнего края виджета до базовой линии и тип базовой линии соответственно.
* Center - виджет, который центрирует своего ребенка внутри себя. Он может также изменять свой размер в зависимости от ребенка, если задано свойство widthFactor или heightFactor.
* ConstrainedBox - виджет, который накладывает дополнительные ограничения на своего ребенка. Ограничения определяются свойством constraints, которое принимает объект BoxConstraints, содержащий минимальную и максимальную ширину и высоту.
* Container - удобный виджет, который объединяет в себе общие свойства рисования, позиционирования и изменения размеров виджетов. Он может иметь цвет, границу, тень, отступ, заполнение, ограничения, выравнивание, трансформацию и декорацию.
* MediaQuery - виджет, который предоставляет информацию об окружении медиа, такую как размер, ориентация, яркость и контрастность экрана. Он использует объект MediaQueryData, который можно получить с помощью статического метода MediaQuery.of(context).

Если вы хотите узнать больше о Single-child layout Widgets в Flutter, вы можете посетить руководство по языку Flutter или статью о виджетах компоновки.

.

16)Определение и особенности Single-child layout Widgets в Flutter. Виджеты: Expanded, Padding, SizedBox, Transform, OverflowBox, FittedBox.

Single-child layout Widgets в Flutter - это виджеты, которые принимают один дочерний виджет и управляют его размером, позицией или другими аспектами компоновки. Они полезны для создания сложных интерфейсов из простых элементов[1](https://education.yandex.ru/handbook/flutter/article/widgets-layout)[2](https://docs.flutter.dev/ui/widgets/layout). Вот некоторые из них:

Expanded - разворачивает дочерний элемент Row/Colum/Flex, таки образом, чтобы дочерний элемент заполнял доступное пространство.

Padding - добавляет отступы вокруг своего дочернего виджета.

SizedBox - виджет позволяет задать фиксированный размер своего дочернего виджета. Эти значения будут проигнорированы, если родитель SizedBox не разрешает их.

Transform - виджет позволяет применить преобразования, такие как масштабирование, поворот и смещение. перед рисованием своего дочернего элемента!

OverflowBox - виджет накладывает на свой дочерний элемент ограничения, отличные от получаемых от своего родителя, возможно, позволяя дочернему элементу переполнять родительский. У родителя OverflowBox размер 10х10, а у ребенка может быть 20х20.

FittedBox виджет масштабирует своего дочернего виджета, чтобы он соответствовал доступному пространству. Он позволяет масштабировать содержимое, сохраняя его пропорции.

17)Определение и особенности Multi-child layout Widgets в Flutter. Виджеты: Column, GridView, Stack, IndexedStack.

Это виджеты, которые могут содержать несколько дочерних виджетов и предоставляют возможности для управления их размещением и размерами в пределах доступного пространства.

Column - Виджет, отображающий свои дочерние элементы в вертикальном массиве.

GridView - Прокручиваемый 2D-массив виджетов. Можно выбрать куда будет расширяться ([scrollDirection](https://api.flutter.dev/flutter/widgets/ScrollView/scrollDirection.html)) и crossAxisCount сколько будет элементов в перпендикулярном направлении от расширения.

Stack - Виджет, который размещает свои дочерние элементы относительно краев своего блока.

Этот класс полезен, если вы хотите перекрывать несколько дочерних элементов

IndexedStack - отображает один элемент из списка дочерей == указанному индексу.

18)Определение и особенности Multi-child layout Widgets в Flutter. Виджеты: Row, ListView, ListTile, Table, Wrap.

Это виджеты, которые могут содержать несколько дочерних виджетов и предоставляют возможности для управления их размещением и размерами в пределах доступного пространства.

row - Виджет, отображающий свои дочерние элементы в горизонтальном массиве.

list Tile - Прокручиваемый список виджетов, расположенных линейно.

ListTile - Одна строка фиксированной высоты

table - Виджет, использующий алгоритм компоновки таблиц для своих дочерних элементов. в начале можно указать параметры для столбцов, а потом через кучу TableRow() заполнять строки.

wrap - Виджет, отображающий дочерние элементы в нескольких горизонтальных или вертикальных последовательностях.

19)Определение и особенности Component Widgets в Flutter. Виджеты: AppBar, RaisedButton, Scaffold, FloatingActionButton.

Component Widgets в Flutter - это предопределенные виджеты, которые предоставляют готовые компоненты для создания пользовательского интерфейса в приложениях. Они представляют собой готовые строительные блоки, которые можно использовать для создания различных элементов интерфейса.

AppBar - это виджет, который представляет верхнюю панель приложения (панель приложения). Он обычно содержит заголовок, кнопки навигации и другие элементы управления. Особенности AppBar включают возможность настройки заголовка, фона, кнопок и других элементов, а также интеграцию с панелью навигации.

RaisedButton - это виджет, представляющий собой поднятую кнопку на экране. Он реагирует на нажатия пользователей и может выполнять определенные действия. Особенности RaisedButton включают возможность настройки текста, цвета фона, цвета текста, а также обработки событий нажатия.

Scaffold - это виджет, представляющий базовый макет приложения. Он содержит основные элементы интерфейса, такие как AppBar, тело и дополнительные элементы. Особенности Scaffold включают интеграцию с AppBar, возможность добавления различных виджетов в тело приложения, настройку темы, обработку бокового меню и другие функции.

FloatingActionButton - это виджет, представляющий плавающую кнопку действия. Он обычно располагается над основным контентом приложения

20)Определение и особенности Component Widgets в Flutter. Виджеты: Drawer, PopupButton, Date&Time PickerDialog.

Component Widgets в Flutter - это виджеты, которые представляют собой готовые компоненты интерфейса, реализующие стандарты Material Design. Они позволяют создавать красивые и функциональные приложения с минимальным кодом.

Drawer - это виджет, представляющий боковую выдвижную панель, которая обычно содержит навигационные элементы и другие вспомогательные функции.

PopupMenuButton - это виджет, представляющий всплывающее меню с вариантами выбора.

Date&Time PickerDialog (выбор даты и времени) - это виджет, представляющий диалоговое окно для выбора даты и/или времени. Он позволяет пользователю выбирать даты и время с помощью визуального интерфейса.

21)Определение и особенности Component Widgets в Flutter. Виджеты: AlertDialog, BottomSheet, ExpansionPanel, SimpleDialog, SnackBar.

Визуальные, поведенческие и динамичные виджеты

Component Widgets в Flutter - это виджеты, которые представляют собой готовые компоненты интерфейса, реализующие стандарты Material Design. Они позволяют создавать красивые и функциональные приложения с минимальным кодом.

AlertDialog - это виджет, представляющий диалоговое окно, которое отображает сообщение и опциональные кнопки. Он используется для вывода важных сообщений или запроса подтверждения от пользователя. Особенности AlertDialog включают возможность настройки заголовка, BottomSheet - содержимого, кнопок и обработки событий нажатия на кнопки.

это виджет, представляющий раскрывающуюся панель, которая отображается снизу экрана.

ExpansionPanel Панель расширения материала. Он имеет заголовок и тело и может быть либо развернутый или свернутый. Тело панели видно только тогда, когда оно расширенный.

SimpleDialog - это виджет, представляющий простое диалоговое окно с вариантами выбора.

SnackBar - это виджет, представляющий временное всплывающее уведомление, которое отображается в нижней части экрана. Буквально всплывающее окно об успешной операции снизу.

22)Определение и особенности Component Widgets в Flutter. Виджеты: Card, Divider, Stepper, Chip.

Component Widgets в Flutter - это виджеты, которые представляют собой готовые компоненты интерфейса, реализующие стандарты Material Design. Они позволяют создавать красивые и функциональные приложения с минимальным кодом.

Card - это виджет, представляющий карточку, которая содержит контент и может иметь тень и скругленные углы. Он обычно используется для отображения информации в упорядоченном формате, таком как списки или информационные блоки.

Divider - это виджет, представляющий горизонтальную линию, которая используется для разделения различных разделов или элементов интерфейса

Stepper - это виджет, представляющий последовательность шагов или этапов, которые пользователь должен выполнить.

Chip - это виджет, представляющий компактный элемент, который представляет собой некоторое выбранное значение или категорию. Он обычно используется для отображения тегов, фильтров или выбранных элементов.

23)Взаимодействие виджетов в Flutter с пользователем. Виджеты: Dismissible, DragTarget, Draggable, GestureDetector, LongPressDraggable.

Взаимодействие виджетов с пользователем в Flutter возможно благодаря различным виджетам, которые обеспечивают обработку жестов, перетаскивание и другие действия.

Dismissible - это виджет, который позволяет пользователю отклонить (удалить) элемент интерфейса путем свайпа в определенном направлении. Он обычно используется для удаления элементов из списка или других аналогичных действий.

DragTarget - это виджет, который определяет область, в которую можно перетащить другой виджет (Draggable).

Draggable - это виджет, который позволяет пользователю перетаскивать элемент интерфейса по экрану. Используется с DragTarget.

GestureDetector - это виджет, который обрабатывает различные жесты и события, совершаемые пользователем, такие как нажатие, двойное нажатие, свайп и т. д. Он обычно используется для добавления интерактивности к другим виджетам. Особенности GestureDetector включают возможность настройки обработчиков событий для различных жестов и атрибутов, таких как время задержки и радиус нажатия.

LongPressDraggable - это виджет, который позволяет пользователю длительно нажимать на элемент интерфейса, чтобы начать перетаскивание. Он подобен Draggable, но требует длительного нажатия для начала перетаскивания.

24)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoActionSheet, CupertinoSegmentedControl, CupertinoPicker.

Библиотека Cupertino в Flutter является частью фреймворка, которая предоставляет виджеты и стили, характерные для операционной системы iOS. Она моделирует внешний вид и поведение компонентов пользовательского интерфейса, которые обычно используются в приложениях для iOS.

CupertinoActionSheet - это виджет, который отображает диалоговое окно с действиями для выбора пользователем. Он обычно используется для отображения списка действий или вариантов выбора в стиле, характерном для iOS. Особенности CupertinoActionSheet включают возможность настройки заголовка, сообщения, действий и визуального оформления.

CupertinoSegmentedControl - это виджет, который представляет собой горизонтальный или вертикальный сегментированный контроль, позволяющий пользователю выбирать один из нескольких вариантов. Он обычно используется для предоставления набора взаимоисключающих вариантов выбора. Подойдет для выбора пола.

CupertinoPicker отображает дочерние виджеты на колесике для выбора и выполняет обратный вызов при изменении текущего выбранного элемента.

25)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoPageTransition, CupertinoPageScaffold, CupertinoNavigationBar.

Библиотека Cupertino в Flutter является частью фреймворка, которая предоставляет виджеты и стили, характерные для операционной системы iOS. Она моделирует внешний вид и поведение компонентов пользовательского интерфейса, которые обычно используются в приложениях для iOS.

CupertinoPageTransition - это виджет, который обеспечивает анимацию при переходе между страницами в приложении. Он обычно используется для создания плавных и анимированных переходов между экранами.

CupertinoPageScaffold - это виджет, который представляет собой базовый макет страницы в стиле iOS. Он обычно используется в качестве основы для создания экранов в приложении.

CupertinoNavigationBar - это виджет, который представляет собой навигационную панель в стиле iOS. Он обычно используется вместе с CupertinoPageScaffold для создания единого внешнего вида приложения. CupertinoNavigationBar позволяет задавать заголовок, кнопки навигации (например, кнопку "Назад") и другие элементы.

26)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoDialog, CupertinoButton, CupertinoContextMenu, CupertinoDatePicker, CupertinoTimerPicker.

Библиотека Cupertino в Flutter является частью фреймворка, которая предоставляет виджеты и стили, характерные для операционной системы iOS. Она моделирует внешний вид и поведение компонентов пользовательского интерфейса, которые обычно используются в приложениях для iOS.

CupertinoDialog - это виджет, который представляет собой диалоговое окно в стиле iOS. Он обычно используется для отображения важной информации, предупреждений или запросов подтверждения действий от пользователя.

CupertinoButton - это виджет, представляющий кнопку в стиле iOS.

CupertinoContextMenu - это виджет, который представляет собой контекстное меню в стиле iOS. Он обычно используется для отображения списка вариантов или действий, доступных для элемента интерфейса. Допустим нажимаешь на картинку и под картинкой вылазит изменить, сохранить, удалить и т.п.

CupertinoDatePicker - это виджет, который представляет собой компонент выбора даты и времени в стиле iOS. Он обычно используется для позволяет пользователю выбирать дату или время из предоставленных вариантов. CupertinoDatePicker позволяет настраивать тип выбора (дата, время или оба), начальное значение и обработчики событий.

CupertinoTimerPicker - это виджет, который представляет собой компонент выбора времени с таймером в стиле iOS. Он обычно используется для выбора временного интервала или установки таймеров. CupertinoTimerPicker позволяет настраивать начальное значение, шаги выбора времени и обработчики событий.

27)Понятие и назначение библиотеки Cupertino в Flutter. Виджеты: CupertinoTabScaffold, CupertinoTabView, CupertinoTextField, CupertinoButton.

Библиотека Cupertino в Flutter является частью фреймворка, которая предоставляет виджеты и стили, характерные для операционной системы iOS. Она моделирует внешний вид и поведение компонентов пользовательского интерфейса, которые обычно используются в приложениях для iOS.

CupertinoTabScaffold - это виджет, который представляет собой основу для создания макета с вкладками в стиле iOS. Он обычно используется в приложениях с несколькими экранами, где каждая вкладка представляет собой отдельный контент. CupertinoTabScaffold включает в себя навигационную панель вкладок и область для отображения содержимого каждой вкладки.

CupertinoTabView - это виджет, который представляет собой содержимое вкладки в макете с вкладками. Он обычно используется в паре с CupertinoTabScaffold. CupertinoTabView позволяет разработчикам определить содержимое и поведение каждой вкладки в приложении.

CupertinoTextField - это виджет, который представляет собой поле ввода текста в стиле iOS. Он обычно используется для получения ввода текста от пользователя.

CupertinoButton - это виджет, который представляет собой кнопку в стиле iOS.

28)Навигация в приложении Flutter. Определение и назначение Navigator, MaterialPageRoute, CupertinoPageRoute.

В приложениях Flutter навигация отвечает за переход между различными экранами и управление стеком маршрутов. Navigator является ключевым классом для управления навигацией в Flutter.

Navigator - это класс в Flutter, который управляет стеком маршрутов и обеспечивает навигацию между экранами. Он предоставляет методы для перехода на новый экран, возврата на предыдущий экран, замены текущего экрана и других навигационных операций. Navigator работает с маршрутами, которые определяют, какой виджет должен быть отображен на экране.

MaterialPageRoute - это класс, который представляет собой реализацию маршрута (Route) в стиле Material Design. MaterialPageRoute предоставляет анимацию перехода между экранами, обеспечивает возможность передачи данных между экранами и автоматически добавляет стандартную навигационную панель с кнопкой "Назад".

CupertinoPageRoute - это класс, который представляет собой реализацию маршрута (Route) в стиле Cupertino (iOS). Он используется вместе с Navigator для создания переходов между экранами в Flutter-приложении с использованием визуального стиля и анимации, характерных для iOS. CupertinoPageRoute предоставляет анимацию перехода, поддержку свайпа для возврата на предыдущий экран и другие особенности, соответствующие дизайну iOS.

29)Понятие Flutter Platform Channel. Процесс создания и способы применения.

Flutter Platform Channel - это механизм в Flutter, который позволяет взаимодействовать с кодом на нативных платформах, таких как Android и iOS. Он предоставляет возможность обмена данными и вызова методов между кодом Flutter и кодом на нативной платформе.

Процесс создания и использования Flutter Platform Channel включает следующие шаги:

1. Определение API на нативной платформе:  
   Сначала необходимо определить API на нативных платформах (например, Java/Kotlin для Android или Objective-C/Swift для iOS), которое будет использоваться для взаимодействия с кодом Flutter. API может включать методы и события, которые вы хотите сделать доступными для использования в коде Flutter.
2. Создание Flutter Platform Channel:  
   В коде Flutter нужно создать экземпляр класса MethodChannel или EventChannel из пакета flutter/services. MethodChannel используется для вызова методов на нативной платформе, а EventChannel - для передачи событий из нативной платформы в код Flutter.
3. Реализация обработчиков на нативной платформе:  
   В коде на нативной платформе нужно реализовать обработчики для методов и событий, определенных на шаге 1. Когда метод вызывается из кода Flutter через Flutter Platform Channel, соответствующий обработчик вызывается на нативной платформе для выполнения соответствующей операции.
4. Использование Flutter Platform Channel:  
   Теперь можно использовать Flutter Platform Channel для вызова методов на нативной платформе или получения событий из нативного кода. В коде Flutter нужно вызвать методы invokeMethod() для отправки запросов на нативную платформу или использовать Stream для получения событий из нативного кода.

Применение Flutter Platform Channel может быть полезно, когда вам нужно взаимодействовать с функциональностью, которая доступна только на нативных платформах, например, доступ к специфическим API устройства, низкоуровневым функциям или сторонним библиотекам.

30)Алгоритм работы с файловой системой в Flutter.

Импорт пакета path:

Для работы с путями файлов и директорий рекомендуется импортировать пакет path, который предоставляет удобные методы для работы с путями в кросс-платформенном контексте.

import ‘package:path/path.dart’ as path;

Получение пути к директории приложения:

Для доступа к файловой системе нужно получить путь к директории, в которой выполняется приложение. Можно использовать метод getApplicationDocumentsDirectory() из пакета path\_provider для получения пути к директории документов:

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

Future<String> getApplicationDirectoryPath() async {

final directory = await pathProvider.getApplicationDocumentsDirectory();

return directory.path;

}

Для создания, чтения и записи файлов можно использовать классы File и Directory из пакета dart:io. File представляет отдельный файл, а Directory представляет директорию.

Создание файла:

final filePath = path.join(directoryPath, 'file.txt');

final file = File(filePath);

await file.create();

Чтение из файла:

final contents = await file.readAsString();

Запись в файл:

final content = 'Hello, World!';

await file.writeAsString(content);

31)Назначение библиотек Sqflite и SharedPreference в Flutter.

Sqflite - библиотека для работы с базами данных SQLite в приложениях Flutter. Она предоставляет удобный API для создания, чтения, записи и выполнения запросов к базе данных SQLite. Sqflite позволяет сохранять структурированные данные в базе данных на устройстве и обеспечивает механизмы для эффективного управления данными в приложении.

SharedPreference - это простой пакет для работы с хранилищем настроек (preferences) в Flutter. Он предоставляет удобные методы для сохранения и получения данных в виде пар ключ-значение. SharedPreference используется для хранения небольших объемов данных, таких как настройки пользователя, состояние приложения или предпочтения пользователя.

32)Принцип обработки разрешений в приложениях Flutter. Инструменты для управления разрешениями.

Первоначально, приложение должно запросить соответствующие разрешения у пользователя. Для этого можно использовать пакет permission\_handler, который предоставляет удобный API для запроса разрешений в Flutter.

Permission.camera.request();

После запроса разрешений, пользователь может предоставить или отклонить их. Вы можете обработать ответ пользователя в соответствии с вашей логикой приложения.

Так же во время работы приложения может потребоваться проверить текущий статус разрешений. Например, если вы хотите выполнить операцию, требующую разрешения.

Permission.camera.status;

* permission\_handler: Этот пакет предоставляет удобный API для запроса и проверки разрешений в Flutter. Он поддерживает различные разрешения, такие как доступ к камере, микрофону, местоположению, контактам и другим системным ресурсам.

33)Механизм отправки и обработки уведомлений в Flutter. Способы и инструменты.

Firebase Cloud Messaging (FCM) — это кроссплатформенное решение для обмена сообщениями, которое позволяет надежно отправлять сообщения бесплатно.

Используя FCM, вы можете уведомить клиентское приложение о том, что новое электронное письмо или другие данные доступны для синхронизации. Вы можете отправлять уведомления, чтобы стимулировать повторное вовлечение и удержание пользователей. В таких случаях использования, как обмен мгновенными сообщениями, сообщение может передавать в клиентское приложение полезную нагрузку размером до 4 КБ.

34)Понятие Firebase. Виды и назначение сервисов Firebase, доступные для интеграции с Flutter.

Firebase - это платформа разработки приложений, предоставляемая Google. Она предлагает различные сервисы и инструменты. Firebase обеспечивает облачные вычисления, базы данных, аналитику, аутентификацию, управление учетными записями пользователей, хостинг, хранение файлов и другие функции, которые облегчают разработку и улучшают пользовательский опыт.

Виды сервисов:

Аутентификация (Authentication): Firebase Authentication.

База данных (Database): Cloud Firestore, Firebase Realtime Database.

Хранилище (Storage): Firebase Storage. модели ключ-значение.

Предоставляет облачное хранилище для загрузки, хранения и управления файлами

Облачные функции: позволяет запускать серверный код без необходимости управления серверной инфраструктурой

Уведомления: позволяет разработчикам отправлять уведомления и сообщения на мобильные устройства

Аналитика: возможность сбора данных о поведении пользователей в приложении, анализа этих данных

Тестирование: предоставляет платформу для автоматического тестирования приложений на различных устройствах и конфигурациях, чтобы проверить их работоспособность и совместимость.

Мониторинг производительности: позволяет отслеживать производительность приложения, мониторить задержки и анализировать проблемы

Отладка сбоев: инструменты для отслеживания и анализа сбоев приложения

Динамические ссылки: позволяет создавать универсальные ссылки, которые могут перенаправлять пользователей на определенные экраны вашего приложения, даже если оно не установлено на устройстве.

Индексация приложения: помогает индексировать и отображать содержимое вашего приложения в результатах поиска Google

Встроенные сообщения

Машинное обучение: предоставляет набор инструментов и API для интеграции машинного обучения в ваши приложения. Он предлагает функции распознавания текста, обнаружения лиц, классификации изображений, распознавания языка и другие возможности машинного обучения.

35)Понятия FutureBuilder и StreamBuilder в Flutter. Принципы работы и примеры использования.

FutureBuilder и StreamBuilder являются виджетами, которые облегчают работу с асинхронными операциями и обновлением пользовательского интерфейса на основе полученных данных.

FutureBuilder используется для построения пользовательского интерфейса на основе результата Future, который представляет асинхронную операцию, такую как загрузка данных из сети или чтение данных из базы данных.

Принцип работы:

Вы определяете Future, который будет выполнять асинхронную операцию.

Вы используете FutureBuilder, который принимает Future и строит пользовательский интерфейс на основе его состояния.

Когда Future выполняется, FutureBuilder вызывает соответствующий метод построения пользовательского интерфейса в зависимости от состояния Future (например, ожидание, выполнено успешно или произошла ошибка).

StreamBuilder используется для построения пользовательского интерфейса на основе данных, полученных из потока (stream). Поток представляет непрерывный поток данных, таких как обновления в реальном времени или события пользователя.

Принцип работы:

Вы определяете Stream, который будет поставлять данные.

Вы используете StreamBuilder, который принимает Stream и строит пользовательский интерфейс на основе полученных данных.

Когда данные поступают в поток, StreamBuilder вызывает соответствующий метод построения пользовательского интерфейса, обновляя его с новыми данными.

Крч, похожи, но одно использует поток, а второе асинхронную операцию.

36)Структура и назначение архитектурного шаблона Bloc в Flutter.

Структура BLoC включает следующие компоненты:

**Business Logic Component (BLoC):**

Отвечает за обработку событий и управление состоянием приложения.

Является независимым от пользовательского интерфейса и платформы компонентом.

Обычно реализуется в виде класса, который содержит логику, связанную с бизнес-процессами и операциями приложения.

**Events (События):**

Представляют действия, которые могут происходить в приложении.

Могут быть инициированы пользователем (например, нажатие кнопки) или внутренними процессами приложения (например, получение данных из сети).

Обычно реализуются в виде классов или перечислений.

**States (Состояния):**

Отражают текущее состояние приложения в ответ на события.

Могут быть изменяемыми и отображать различные сценарии приложения, такие как загрузка данных, отображение ошибок или успешное выполнение операции.

Обычно реализуются в виде классов или перечислений.

**User Interface (Пользовательский интерфейс):**

Отображает данные и состояния, предоставляемые BLoC.

Подписывается на состояния BLoC и обновляет себя в соответствии с изменениями состояния.

Может отправлять события BLoC для инициирования новых действий.

**Назначение BLoC:**

Разделение логики приложения и пользовательского интерфейса для повышения читаемости, тестируемости и повторного использования кода.

Управление состоянием и обработка событий в приложении.

Обеспечение единого источника истины для данных, которые могут быть использованы различными виджетами в пользовательском интерфейсе.

Упрощение отладки и обнаружение ошибок, благодаря централизованному управлению состоянием.

37)Преимущества использования пакета Provider в Flutter. Применение Provider для управления состоянием.

**Provider в Flutter** предоставляет простой и эффективный способ передачи и обновления данных между различными виджетами в дереве виджетов без необходимости явного использования колбэков или стратегий управления состоянием.

Преимущества:

-Простота использования. простой и интуитивно понятный API для управления состоянием

-Внедрение зависимостей. позволяет эффективно управлять зависимостями между различными компонентами приложения. Это способствует улучшению модульности и повторному использованию кода.

-Реактивность и автоматическое обновление. Виджеты могут автоматически обновляться, когда изменяются данные, на которые они подписаны.

-Локализованное обновление. Только те виджеты, которые действительно зависят от измененных данных, будут обновлены. Это помогает оптимизировать производительность приложения и уменьшить избыточные перестроения виджетов.

-Расширяемость.

Применение Provider для управления состоянием:

Создание класса модели (model), который содержит данные и методы для их обновления.

Использование Provider для предоставления модели в дереве виджетов приложения.

Подписка виджетов на изменения данных, используя Consumer или Provider.of.

Обновление состояния модели и автоматическое обновление виджетов, которые зависят от этих данных.

38)Принцип использования библиотеки Hive для локального хранения данных в приложениях Flutter.

Hive - это быстрая и легковесная база данных ключ-значение (key-value), разработанная для использования в приложениях Flutter. Она предоставляет простой и эффективный способ локального хранения данных на устройстве пользователя.

**Принципы использования библиотеки:**

Определение моделей данных: Создайте классы моделей данных, которые будут сохраняться в Hive. Эти классы должны реализовывать абстрактный класс HiveObject или быть аннотированы с @HiveType(). Определите поля моделей данных, которые будут сохраняться в Hive, используя аннотации @HiveField().

Инициализация Hive: В начале вашего приложения, обычно в методе main(), инициализируйте Hive с помощью await Hive.initFlutter(). Вы можете настроить дополнительные параметры, такие как путь к каталогу базы данных или шифрование данных.

Открытие боксов (контейнеров): Бокс - это контейнер в Hive, в котором хранятся объекты моделей данных. Создайте или откройте бокс с помощью Box box = await Hive.openBox('myBox'), где 'myBox' - это имя бокса.

Добавление и получение данных: Добавьте данные в бокс с помощью метода box.add(). Получите данные из бокса с помощью методов box.get(), box.getAt(), box.values, box.keys и других.

Обновление и удаление данных: Обновите данные в боксе, изменяя объект модели данных и вызывая box.put(). Удалите данные с помощью методов box.delete(), box.deleteAt() или box.clear() для удаления всех данных в боксе.

Закрытие боксов и завершение работы: После завершения работы с боксами, закройте их с помощью box.close(). При завершении работы приложения, вызовите Hive.close() для закрытия Hive.

39)Определение и назначение юнит-тестов в Flutter. Роль библиотеки Mockito в тестировании.

Юнит-тесты в Flutter представляют собой автоматизированные тесты, которые проверяют отдельные части кода (юниты) на предмет соответствия ожидаемому поведению. Целью юнит-тестов является проверка правильности работы отдельных модулей, функций или классов без зависимости от других компонентов системы.

Роль Mockito в тестировании заключается в следующем:

Создание поддельных объектов: Mockito позволяет создавать поддельные объекты (mock-объекты), которые заменяют реальные зависимости в юнит-тестах. Это позволяет изолировать тестируемый модуль от его зависимостей и фокусироваться на проверке конкретного поведения.

Определение поведения поддельных объектов: Mockito предоставляет методы для настройки поведения поддельных объектов. Вы можете указать, какие значения должны быть возвращены при вызове определенных методов или какие исключения должны быть брошены. Это позволяет смоделировать различные сценарии и проверить, как тестируемый модуль взаимодействует с зависимостями.

Проверка вызовов методов: Mockito предоставляет методы для проверки, были ли вызваны определенные методы на поддельных объектах и с какими аргументами. Это позволяет убедиться, что код внутри тестируемого модуля взаимодействует с его зависимостями правильно.

Упрощение тестирования: Благодаря Mockito можно создавать компактные и легко поддерживаемые юнит-тесты. Она предоставляет удобные методы и синтаксический сахар для настройки поддельных объектов и проверки вызовов методов.

40)Тестирование виджетов в Flutter. Методы поиска и проверки конкретных виджетов в приложении.

Для поиска виджетов можно использовать find и вызывать у него различные методы по типу byType(тип виджета), byKey(уникальный ключ), text(текст), descendant(потомок).

Проверка через expect. Можоно проверять буквально все.

Свойства виджета: текст, цвет, размер шрифта и т.п.

Дочерние элементы: наличие, порядок, и их свойства.

Проверка взаимодействия с виджетом: нажатие на кнопки, дождаться реакции, проверить, что реакция является той, что нам нужна.

Изменение состояние.

41)Интеграционное тестирование в приложениях Flutter. Инструменты для проверки взаимодействия компонентов.

Интеграционное тестирование в приложениях Flutter - это процесс тестирования, в котором проверяется взаимодействие различных компонентов приложения вместе. Оно помогает обнаруживать проблемы, которые могут возникнуть при интеграции отдельных модулей приложения. Для проверки взаимодействия компонентов в интеграционных тестах можно использовать следующие инструменты:

инструмент Flutter Driver

пакет IntegrationTest

класс WidgetTester ?