Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

А. А. Сидоров
Р. С. Кульшин

РАЗРАБОТКА МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

Методические указания к лабораторным работам
и организации самостоятельной работы для студентов направления
«Программная инженерия»
(уровень бакалавриата)

Томск
2018

Сидоров, Анатолий Анатольевич
С34 Разработка мобильных приложений: методические указания к лаборатор-ным работам и организации самостоятельной работы для студентов направления «Программная инженерия» (уровень бакалавриата) / А. А. Сидоров, Р. С. Куль-шин. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2018. – 36 с.

Методические указания предназначены для сопровождения образовательного процесса по дисциплине «Разработка мобильных приложений» в части подготовки обучающихся к ла-бораторным работам и организации их самостоятельной работы.

Для студентов высших учебных заведений, обучающихся по направлению «Программ-ная инженерия» (уровень бакалавриата), а также иным направлениям и специальностям, предусматривающим освоение цифровых компетенций в области создания мобильных прило-жений.

Оглавление

ВВЕДЕНИЕ

Выполнение лабораторных работ и самостоятельная работа направлены на приобретение навыков разработки приложения для устройств работающих на базе мобильных операционных систем Android и iOS, технологий и инструментов проектирования и разработки мобильных приложений, изучение кроссплатформенного фреймворка мобильной разработки – Flutter и интегрированной среды разработки Android Studio студентами направления подготовки бакалавров «Программная инженерия».

В результате выполнения лабораторных работ и осуществления самостоятельной работы обучающийся должен достичь следующих результатов:

– знать историю мобильных операционных систем; основные подходы к разработке мобильных приложений; принципы работы фреймворка Flutter; основные виджеты мобильных приложений и средства их компоновки; принципы взаимодействия с пользователем; основы управления состояниями; файловую систему мобильных операционных систем; средства долговременного хранения данных; архитектуры проектирования мобильных приложений; режимы сборки мобильных приложений; правила подготовки мобильных приложений к релизу;

– уметь применять стандартные виджеты фреймворка Flutter; разрабатывать собственные виджеты; взаимодействовать с сетью; использовать средства долговременного хранения данных; управлять состояниями мобильного приложения; выстраивать канальную связь; проектировать архитектуру мобильных приложений разной степени сложности; подготавливать приложение к релизу;

– владеть языком программирования Dart; фреймворком Flutter; интегрированной средой разработки Android Studio; методами разработки мобильных приложений; навыками исследования современных операционных систем и технологий разработки.

В ходе освоения дисциплины у студента должны быть сформированы цифровые компетенции и профессиональное мышление.

1 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРОВЕДЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

1.1 Лабораторная работа «Запуск базового проекта и использование основных виджетов»

Цель работы

Научиться устанавливать средства разработки мобильных приложений, изучить структуру проекта Flutter и использовать основные виджеты.

Форма проведения

Выполнение индивидуального задания

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

Flutter представляет фреймворк от компании Google, который позволяет создавать кроссплатформенные приложения, которые могут использовать один и тот же код. Спектр платформ широк – это веб-приложения, мобильные приложения под Android и iOS, графические приложения под настольные операционные системы Windows, MacOS, Linux, а также веб-приложения.

Особенностью работы с Flutter является то, что приложения под разные платформы могут иметь один и тот же код. Поскольку используемые платформы не эквиваленты, то какие-то отдельные части кода необходимо настраивать под определенную ОС, например, под iOS, но, тем не менее, большая часть кода может совпадать. Это позволяет разработчикам существенно сэкономить время и ресурсы на создание приложений под все поддерживаемые платформы.

В качестве языка разработки используется язык программирования Dart.

При построении приложения Flutter транслирует код на Dart в нативный код приложения с помощью Dart AOT (компиляция приложения перед его запуском), которое можно запускать на Android или iOS или другой платформе. Однако при разработке приложения для ее ускорения Flutter использует JIT (компиляция приложения в процессе его запуска).

Надо отметить, что Flutter относительно новый фреймворк. Хотя прототип фреймворка появился еще в 2015 году, а первая альфа-версия вышла в мае 2017 года, но первый стабильный релиз – Flutter 1.0 – увидел свет только в декабре 2018 года. Тем не менее, регулярно с небольшой периодичностью выходят версии, добавляется новая функциональноть и исправляются имеющиеся баги.

Для написания кода программы можно использовать любой текстовый редактор. Затем с помощью утилит командной строки из Flutter SDK компилировать приложение. Однако для таких сред как Android Studio и IntelliJ IDEA, а также текстового редактора Visual Studio Code компания Google выпустила специальные плагины, которые позволяют упростить разработку. Поэтому зачастую для разработки под Flutter используются именно Android Studio.

Основные элементы структуры проекта:

– Папка .dart-tool – специальная папка, которая хранит информацию об используемых пакетах.

– Папка .idea – специальная папка для Android Studio, которая содержит базовую конфигурацию.

– Папка android содержит код и дополнительные файлы, которые позволяют связать приложение на Dart с Android.

– Папка ios содержит код и дополнительные файлы, которые позволяют связать приложение на Dart с iOS.

– Папка buid содержит файлы, создаваемые в результате процесса построения приложения.

– Папка lib содержит собственно файлы приложения на языке Dart.

– Папка test предназначена для хранения файлов с тестами.

– Папка web содержит код и дополнительные файлы для создания веб-приложения на Flutter.

– Файл pubsec.yaml хранит конфигурацию проекта, в частности, пакет проекта, список зависимостей и т.д.

Порядок выполнения работы

Установка Flutter и Android Studio

Для установки Flutter SDK перейдем на страницу https://flutter.dev/docs/get-started/install/windows. На этой странице найдем в секции ***Get the Flutter SDK*** ссылку на zip-архив с Flutter SDK и загрузим его.

Далее распакуем архив, например, на диске C. В распакованном архиве в папке flutter/bin мы найдем инструменты для компиляции приложения (рисунок 1).

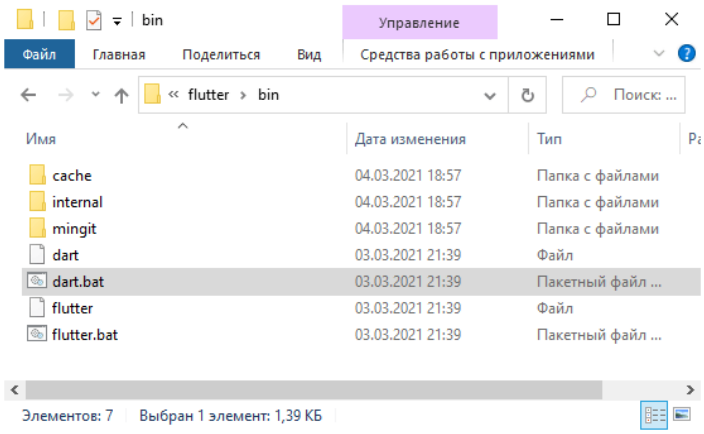


Рисунок 1 – Папка Flutter SDK

Если работаем в ОС Windows, то для добавления переменной среды через поиск найдем параметр «Изменение переменных среды текущего пользователя». Для этого введем в поле поиска «Изменение переменных». Выберем пункт «Изменение переменных среды текущего пользователя». Затем откроется окно, где можно увидеть все переменные среды. (Также можно перейти через Параметры и пункт Система -> Дополнительные параметры системы -> Переменные среды). Здесь нужно изменить переменную Path, добавив путь к папке bin в Flutter SDK. Для этого выберем пункт Path и нажмем на кнопку «Изменить». Далее нажмем на кнопку «Создать» и появившееся поле ввода введем путь к папке bin из Flutter SDK.

Чтобы проверить корректность установки Flutter, откроем командную строку и введем команду flutter. Если Windows распознает ее, и последует вывод некоторой справочной информации (например, как использовать те или иные команды в консоли), то flutter установлен и настроен (рисунок 2).

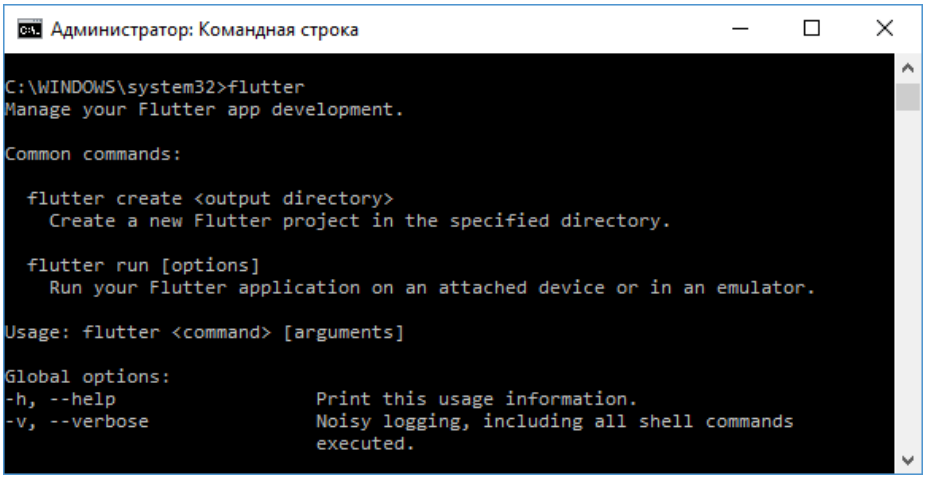


Рисунок 2 – Вывод консоли при корректной установке Flutter SDK

Для установки Android Studio необходимо перейти по ссылке и нажать кнопку скачать https://developer.android.com/studio. Далее необходимо установить Android Studio следуя инструкции установки.

После установки необходимо запустить Android Studio, перейти во вкладку Plugins и найти Flutter, затем установить этот плагин (рисунок 3).

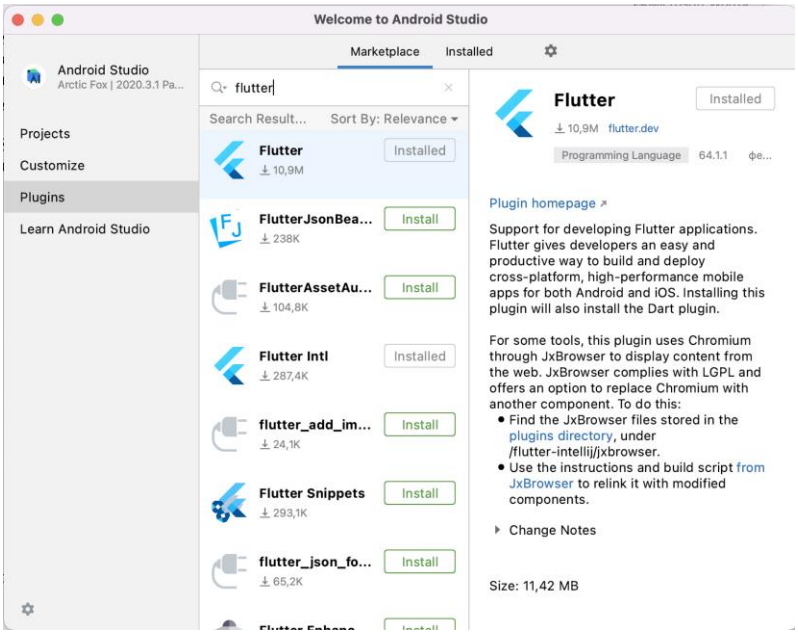


Рисунок 3 – Установка плагина Flutter

Запуск базового проекта

Далее во вкладке Projects нужно нажать New Flutter Project для создания проекта приложения на Flutter.

Для запуска стандартного проекта в верхнем меню выберите устройство и нажмите кнопку запуска (рисунок 4).

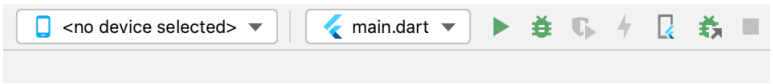


Рисунок 4 – Панель запуска

После запуска приложение скомпилируется и запустится на выбранном устройстве (рисунок 5).

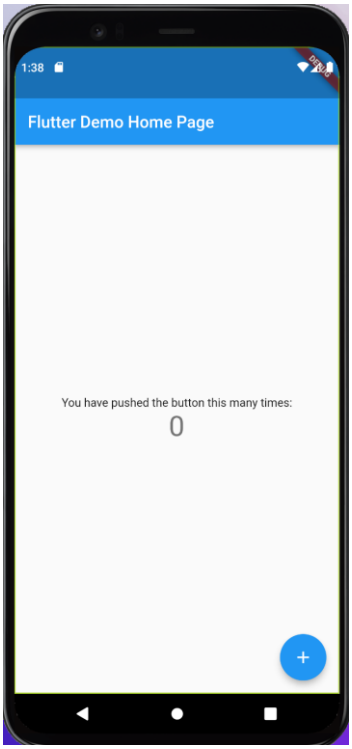


Рисунок 5 – Стандартное приложение Flutter

Далее необходимо очистить файл main.dart и перейти к выполнению задания.

Общим заданием является изучение теоретических материалов, создание базовой структуры, содержащей runApp, MaterialApp, Scaffold и AppBar. В AppBar Вам необходимо указать свое ФИО. После этого шага расширьте свое приложение исходя из индивидуального варианта.

Варианты заданий

1. Добавьте в Leading AppBar произвольную иконку используя класс Icons. В тело экрана добавьте виджет текста содержащий название дисциплины и примените к нему произвольные стили TextStyle.
2. Измените цвет AppBar используя класс Colors. Добавьте в ассеты проекта картинку и выведете ее в тело экрана используя Image.assets.
3. Добавьте в Leading AppBar произвольную иконку используя класс Icons. Добавьте в ассеты проекта картинку и выведете ее в тело экрана используя Image.assets.
4. Измените цвет AppBar используя класс Colors. В тело экрана выведете картинку из интернета используя класс Image.network и задайте ей отступы используя класс Padding.
5. Добавьте в Leading AppBar произвольную иконку используя класс Icons. В тело экрана добавьте контейнер и задайте ему цвет, во внутрь контейнера добавьте отступ используя класс Padding и в него добавьте текстовый виджет, содержащий название дисциплины.
6. Измените цвет AppBar используя класс Colors. В тело экрана добавьте виджет текста содержащий название дисциплины и примените к нему произвольные стили TextStyle.
7. Измените цвет AppBar используя класс Colors. В тело экрана добавьте контейнер и задайте ему цвет, во внутрь контейнера добавьте отступ используя класс Padding и в него добавьте текстовый виджет, содержащий название дисциплины.

8. Добавьте в Leading AppBar произвольную иконку используя класс Icons и измените цвет AppBar. В тело экрана добавьте виджет текста содержащий название дисциплины и примените к нему произвольные стили TextStyle.

9. Измените цвет AppBar используя класс Colors. В тело экрана выведете картинку из интернета используя класс Image.asset и задайте ей отступы используя класс Padding.

10. Добавьте в Leading AppBar произвольную иконку используя класс Icons. В тело экрана добавьте контейнер и задайте ему цвет, во внутрь контейнера добавьте отступ используя класс Padding и в него добавьте картинку из интернета используя класс Image.network.

Контрольные вопросы

1. Что такое Flutter?
2. Какова структура Flutter-проекта?
3. Основные виджеты и функции (runApp, Scaffold, MaterialApp, AppBar, Container, Text, TextStyle, Image, Icon, Padding).

1.2 Лабораторная работа «Компоновка виджетов»

Цель работы

Ознакомиться с принципами компоновки и научиться компоновать стандартные виджеты фреймворка Flutter.

Форма проведения

Выполнение индивидуального задания

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

Виджет ***Column*** (рисунок 6) располагает элементы вертикально, в виде столбика. Его основными параметрами являются:

- ***mainAxisAlignment***: задает выравнивание по вертикали;
- ***mainAxisSize***: задает пространство, занимаемое виджетом по основной оси;
- ***crossAxisAlignment***: задает выравнивание по горизонтали;
- ***children***: набор вложенных элементов.

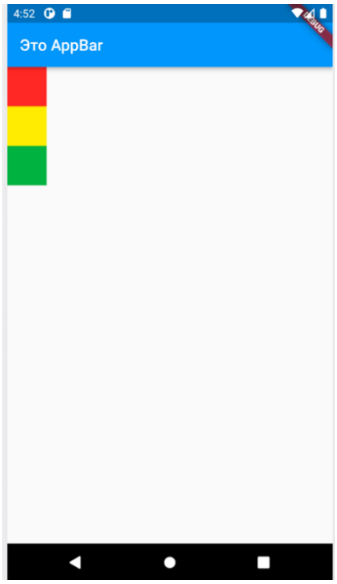


Рисунок 6 – Виджет Column

Виджет ***Row*** (рисунок 7) располагает элементы горизонтально, в виде строки. Его основными параметрами являются:

- ***mainAxisAlignment***: задает выравнивание по горизонтали;
- ***mainAxisSize***: задает пространство, занимаемое виджетом по основной оси;
- ***crossAxisAlignment***: задает выравнивание по вертикали;
- ***children***: набор вложенных элементов.

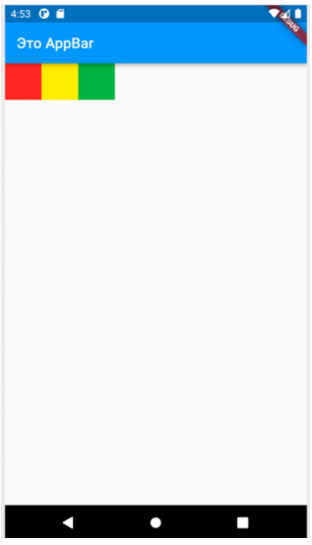


Рисунок 7 – Виджет Row

Виджет ***Stack*** (рисунок 8) позволяет располагать одни элементы поверх других. Основными параметрами являются:

- ***alignment***: задает расположение вложенных виджетов;
- ***textDirection***: определяет порядок расположения вложенных элементов по горизонтали;
- ***fit***: определяет размеры для вложенных виджетов;
- ***children***: набор вложенных элементов.

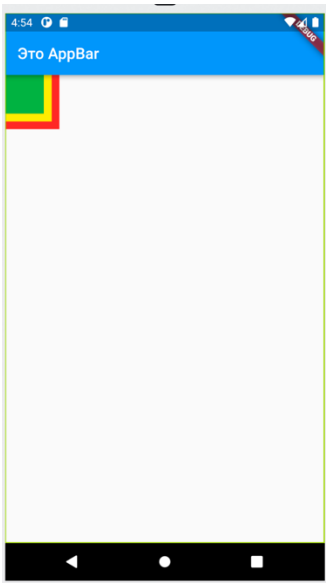


Рисунок 8 – Виджет Stack

Виджет ***Wrap*** (рисунок 9) позволяет располагать элементы в виде сетки.

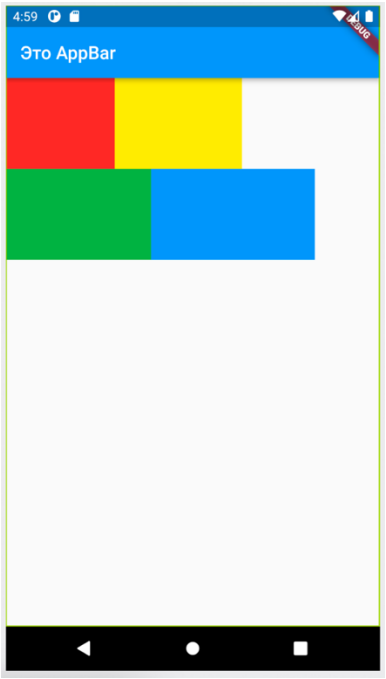


Рисунок 9 – Виджет Wrap

Порядок выполнения работы

1. Изучить теоретические основы.
2. Создать базовый проект.
3. Добавить в приложение AppBar с указанием своего ФИО.
4. Выполнить задание в соответствии своему варианту.
5. Подготовится к контрольным вопросам.

Варианты заданий

1. Добавить в тело экрана виджет Wrap. В виджет Wrap добавить 5-6 контейнеров задать им размер и цвет. Каждому контейнеру задать отступ.
2. Добавить в тело экрана виджет Column. В Column добавить текстовые виджеты - сведения о себе (ФИО, год рождения, группа и т.д.). Каждому текстовому виджету задать отступ и добавить стили TextStyle.
3. Добавить в тело экрана виджет Column. В Column добавить несколько виджетов Row и в каждый Row добавить несколько иконок. Каждому Row добавить выравнивание по ширине.
4. Добавить в тело экрана виджет Stack. В виджет Stack добавить контейнеры разного размера и задать им цвета в соответствии с цветами радуги. Виджет Stack и контейнеры отцентрировать.
5. Добавить в тело экрана виджет Wrap. В виджет Wrap добавить контейнеры, задав им размер и цвет. В каждый контейнер добавить текстовые виджеты – сведения о себе (ФИО, год рождения, группа и т.д.).
6. Добавить в тело экрана виджет Wrap. В виджет Wrap добавить 5–6 контейнеров задать в которых содержаться картинки из интернета. Каждому контейнеру задать отступ.
7. Добавить в тело экрана виджет Column. В виджет Column добавить 5–6 контейнеров задать в которых содержаться картинки из ассетов. Каждому контейнеру задать отступ.
8. Добавить в тело экрана виджет Stack. В виджет Stack добавить несколько контейнеров разного размера и задать им цвета. Все контейнеры необходимо отпозиционировать по углам
9. Добавить в тело экрана виджет Wrap. В виджет Wrap добавить 5–6 контейнеров задать им размер и цвет. Каждому контейнеру задать отступ. В каждый контейнер добавить иконки.
10. Добавить в тело экрана виджет Column. В Column добавить несколько виджетов Row и в каждый Row добавить несколько контейнеров каждый из которых содержит картинки из интернета. Каждому Row добавить выравнивание по ширине.

Контрольные вопросы

1. Виджет Column и его параметры.
2. Виджет Row и его параметры.
3. Виджет Stack и его параметры.
4. Виджет Center.
5. Виджет Wrap.

1.3 Лабораторная работа «Взаимодействие с пользователем и навигация»

Цель работы

Ознакомиться с принципами навигации и научиться использовать инструменты взаимодействия с пользователями.

Форма проведения

Выполнение индивидуального задания

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

Пользовательский ввод

Главным контейнером для формы является виджет-класс ***Form***, он позволяет объединить в себе поля ввода. Обращаясь к состоянию формы FormState, можно проверить корректное заполнение полей, сбросить значения по умолчанию и сохранить значения.

Для ввода текста во Flutter может применяться виджет ***TextFormField***. Реализация виджета TextFormField представлена на рисунке 10.

```
Widget build(BuildContext context) {
  return Container(
    padding: const EdgeInsets.all(10.0),
    child: Form(
      child: Column(
        children: <Widget>[
          const Text(
            'Имя пользователя:',
            style: TextStyle(fontSize: 20.0),
          ),  // Text
          TextFormField(),

        ],  // <Widget>[]
      ),  // Column
    ),  // Form
  );  // Container
}
```

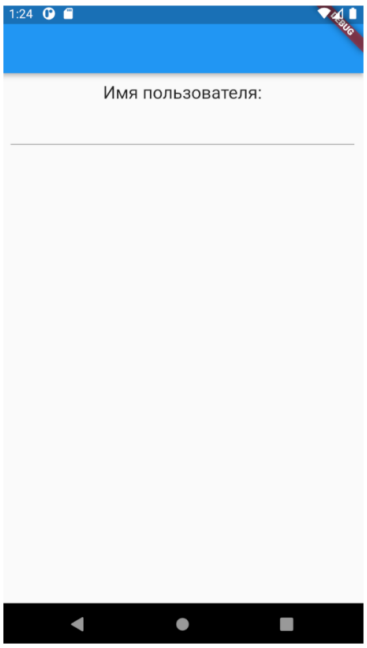
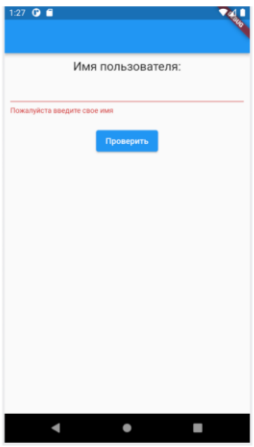


Рисунок 10 – Реализация TextFormField

Пример валидации полей текстового ввода и формы представлен на рисунке 11.




```
final _formKey = GlobalKey<FormState>();

Widget build(BuildContext context) {
  return Container(
    padding: const EdgeInsets.all(10.0),
    child: Form(
      key: _formKey,
      child: Column(
        children: <Widget>[
          const Text(
            'Имя пользователя:',
            style: TextStyle(fontSize: 20.0),
          ),  // Text
          TextFormField(validator: (value) {
            if (value!.isEmpty) return 'Пожалуйста введите свое имя';
          }),  // TextFormField
          const SizedBox(height: 20.0),
          ElevatedButton(
            onPressed: () {
              if (_formKey.currentState!.validate()) {
              }
            },
            child: const Text('Проверить'),
          ),  // ElevatedButton
        ],  // <Widget>[]
      ),  // Column
    ),  // Form
  );  // Container
}
```

Рисунок 11 – Реализация валидации

Для получения данных из полей текстового ввода и дальнейшего использования, полю необходимо задать ключ. Когда необходимо получить данные производим обращение к полю text ключа (рисунок 12).

```dart
class _MyFormState extends State<MyForm> {
  final _formKey = GlobalKey<FormState>();
  final _field = TextEditingController();

  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.all(10.0),
      child: Form(
        key: _formKey,
        child: Column(
          children: <Widget>[
            TextFormField(
              controller: _field,
              validator: (value) {
                if (value!.isEmpty) return 'Пожалуйста введите свое имя';
              },
            ), // TextFormField
            ElevatedButton(
              onPressed: () {
                if (_formKey.currentState!.validate()) {
                  print(_field.text);
                }
              },
              child: const Text('Проверить'),
```

Рисунок 12 – Получение данных из формы

Для TextFormField существует множество форматов ввода, вызывающих разные клавиатуры: числовая, дата, обычная, электронная почта, ссылка и другие. Реализации различных форматов ввода представлены на Рисунке 13.

```dart
TextFormField(
  keyboardType: TextInputType.number,
), // TextFormField
TextFormField(
  keyboardType: TextInputType.emailAddress,
), // TextFormField
TextFormField(
  keyboardType: TextInputType.datetime,
), // TextFormField
TextFormField(
  keyboardType: TextInputType.url,
), // TextFormField
```

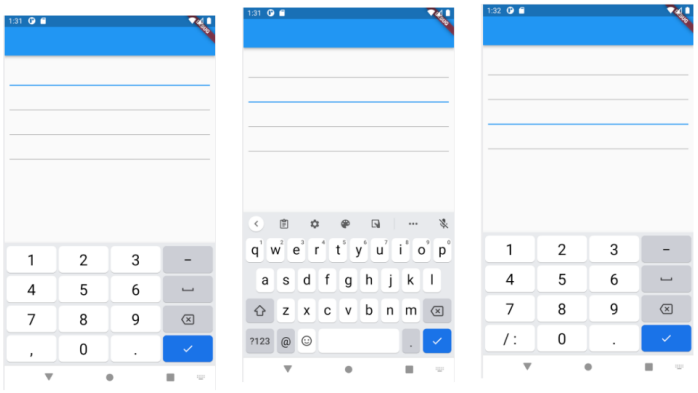
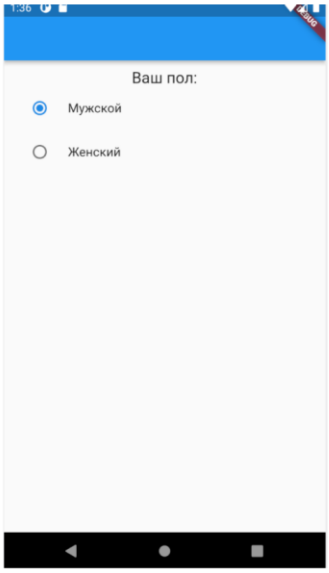


Рисунок 13 – Реализация форматов ввода

Виджет ***RadioListTile*** представляет собой группу радиокнопок, привязанных к групповой переменной. Реализация RadioListTile представлена на рисунке 14.




```dart
class _MyFormState extends State<MyForm> {
  final _formKey = GlobalKey<FormState>();
  int _gender = 1;
  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.all(10.0),
      child: Form(
        key: _formKey,
        child: Column(
          children: <Widget>[
            const Text('Ваш пол:', style: TextStyle(fontSize: 20.0),),
            RadioListTile(
              title: const Text('Мужской'),
              value: 1,
              groupValue: _gender,
              onChanged: (int? value) {setState(() { _gender = value!;});},
            ),  // RadioListTile
            RadioListTile(
              title: const Text('Женский'),
              value: 0,
              groupValue: _gender,
              onChanged: (int? value) {setState(() { _gender = value!;});},
            ),  // RadioListTile

          ],  // <Widget>[]
        ),  // Column
      ),  // Form
    );  // Container
  }
}
```

Рисунок 14 – Реализация RadioListTile

Виджет ***ChekBoxListTile*** представляет группу «флажков» и похожи на радиокнопки, но в отличии от них у каждого чек-бокса имеется своя переменная. На рисунке 15 представлена реализация ChekBoxListTile.

```dart
class _MyFormState extends State<MyForm> {
  final _formKey = GlobalKey<FormState>();
  bool _agreement = false;
  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.all(10.0),
      child: Form(
        key: _formKey,
        child: Column(
          children: <Widget>[
            CheckboxListTile(
                value: _agreement,
                title: new Text('Я ознакомлен с документом "Согласие на обработк
                onChanged: (bool? value) => setState(() => _agreement = value!)
            ),  // CheckboxListTile
          ],  // <Widget>[]
        ),  // Column
      ),  // Form
    );  // Container
  }
}
```

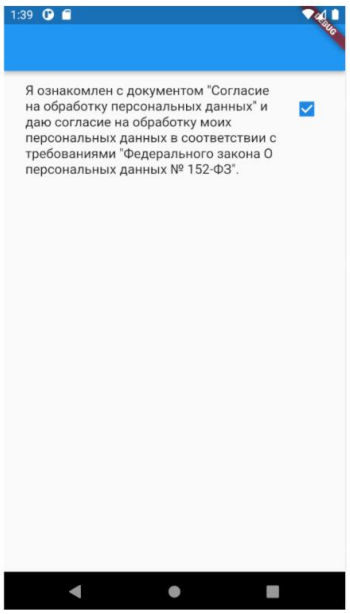


Рисунок 15 – Реализация ChekBoxListTile

Навигация

Navigator – виджет-класс, позволяющий управлять стеком дочерних виджетов, т.е. открывать, закрывать и переключать окна или страницы. При использовании MaterialApp, экземпляр класса Navigator уже создан.

Функции класса навигатора:

– Navigator.push;

– Navigator.pushNamed;
– Navigator.pop и др.

Метод ***push*** позволяет открыть новый экран приложения. При необходимости с его помощью можно передать параметры в новый экран. На рисунке 16 представлена реализация без передачи параметров, а на рисунке 17 с передачей параметров.

```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(context,
                MaterialPageRoute(builder: (context) => SecondScreen()));
          },
          child: Text("Переход на экран"),
        ), // ElevatedButton
      ), // Center
    ); // Scaffold
  }
}
```

```
class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: Text("Второй экран"),
      ), // Center
    ); // Scaffold
  }
}
```

Рисунок 16 – Реализация метода push без передачи параметров

```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) =>
                        SecondScreen(text: "Эти данные я хочу передать")));
          },
          child: Text("Переход на экран"),
        ), // ElevatedButton
      ), // Center
    ); // Scaffold
  }
}
```

```
class SecondScreen extends StatelessWidget {
  String text;
  SecondScreen({required this.text});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: Text(text),
      ), // Center
    ); // Scaffold
  }
}
```

Рисунок 17 – Реализация метода push с передачи параметров

Для того что бы закрыть последний открытый экран используется метод pop. Реализация метода pop представлена на рисунке 18.

```
class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: Text("Закрыть второй экран"),
        ), // ElevatedButton
      ), // Center
    ); // Scaffold
  }
}
```

Рисунок 18 – Реализация метода pop

Порядок выполнения работы

1. Изучить теоретические основы;
2. Создать базовый проект;
3. Добавить в приложение AppBar с указанием своего ФИО;
4. Выполнить задание в соответствии своему варианту;
5. Подготовится к контрольным вопросам;

Варианты заданий

1. Реализовать калькулятор ИМТ. На первом экране должны располагаться 2 числовых поля ввода (вес и рост), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

2. Реализовать калькулятор квадратов суммы. На первом экране должны располагаться 2 числовых поля ввода (число a и число b), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

3. Реализовать калькулятор простых процентов. На первом экране должны располагаться 3 числовых поля ввода (исходный капитал, срок начисления процентов и ставка процентов), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

4. Реализовать калькулятор кинетической энергии. На первом экране должны располагаться 2 числовых поля ввода (масса тела и скорость), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

5. Реализовать калькулятор расчета первой космической скорости. На первом экране должны располагаться 2 числовых поля ввода (масса небесного тела и радиус небесного тела), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

6. Реализовать калькулятор кубов суммы. На первом экране должны располагаться 2 числовых поля ввода (число a, и число b), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

7. Реализовать вычисление корней квадратного уравнения. На первом экране должны располагаться 3 числовых поля ввода (коэффициент a, коэффициент b и коэффициент c), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

8. Реализовать калькулятор ускорения свободного падения. На первом экране должны располагаться 2 числовых поля ввода (масса небесного тела и радиус небесного тела), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

9. Реализовать калькулятор расчета ускорения при равноускоренном движении. На первом экране должны располагаться 3 числовых поля ввода (начальная скорость, конечная скорость и время), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

10. Реализовать калькулятор расчета объема звукового файла. На первом экране должны располагаться 4 числовых поля ввода (тип файла (режим), частота дискретизации, глубина кодирования звука и длительность звуковой дорожки), а также чек-бокс на согласие обработки данных. Реализовать валидацию полей и чек-бокса. Передать данные с первого экрана на второй экран произвести расчеты и отобразить.

Контрольные вопросы

1. Виджет Form.
2. Виджет TextFormFiled.
3. Виджет RadioListTile.
4. Валидация.
5. Ключи формы и полей.
6. Виджет ChekBoxListTile.
7. Класс Navigator.

1.4 Лабораторная работа «Управление состояниями»

Цель работы

Получить понимание управления состояниями и научиться управлять состояниями мобильного приложения.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

Cubit является подмножеством известной реализации шаблона BLoC: bloclibrary.dev, он отказывается от концепции событий и упрощает способ создания состояний. Это класс, который хранит наблюдаемое состояние, наблюдение обеспечивается потоками, но таким дружественным образом, что нет необходимости знать реактивное программирование.

Для примера напишем простой счетчик, в котором можно прибавлять и убавлять значение. Первым делом необходимо добавить flutter_bloc в проект в файл pubspec.yaml (рисунок 19). Актуальную версию библиотеки можно найти на сайте pub.dev.

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.2
  flutter_bloc: ^7.2.0
```

Рисунок 19 – Зависимости проекта

Создадим в проекте папку screens, которая будет содержать провайдер кубита, сам кубит и экран (рисунок 20).

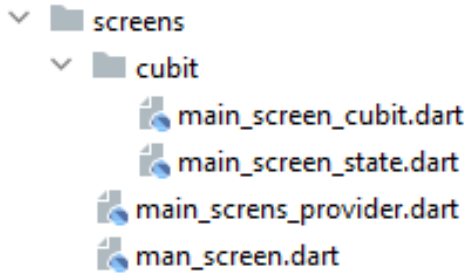


Рисунок 20 – Структура экрана

Теперь создадим классы состояний для нашего кубита. В первую очередь создаем абстрактный класс, от которого будут наследоваться все остальные состояния. После создадим остальные состояния, для нашего счетчика это единственное состояние, изменение его значения (рисунок 21).

```
abstract class MainScreenState{}

class MainScreenUpdateCounterState extends MainScreenState{
  final int value;

  MainScreenUpdateCounterState({required this.value});

}
```

Рисунок 21 – Классы состояния экрана

Далее создадим класс самого кубита (рисунок 22). Создаваемый класс наследуются от базового кубита уже с состоянием, которое прописали выше. Любой кубит должен иметь стартовой состояние, в данном случае это состояние с изменение значения счетчика на 0. Далее напишем две функции: прибавления и убавления значения счетчика (рисунок 23).

```
import 'package:counter_cubit/screens/main_screen/cubit/main_screen_state.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

class MainScreenCubit extends Cubit<MainScreenState>{
  MainScreenCubit() : super(MainScreenUpdateCounterState(value: 0));

}
```

Рисунок 22 – Базовый класс кубита

```dart
import 'package:counter_cubit/screens/main_screen/cubit/main_screen_state.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

class MainScreenCubit extends Cubit<MainScreenState>{
  MainScreenCubit() : super(MainScreenUpdateCounterState(value: 0));

  int counter = 0;

  void addValue(){
    counter++;
    emit(MainScreenUpdateCounterState(value: counter));
  }

  void removeValue(){
    counter--;
    emit(MainScreenUpdateCounterState(value: counter));
  }
}
```

Рисунок 23 – Готовый класс кубита

Теперь создадим сам экран в файле main_screen (рисунок 24). На котором в центре будет само значение счетчика, а также две кнопки для добавления и уменьшения значений счетчика. Для того, чтобы наше значение менялось на экране, в body необходимо добавить ***BlocBuilder***, который будет перестраивать виджет в зависимости от изменения состояний.

```dart
import class MainScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter Cubit'),
        centerTitle: true,
      ),
      body: BlocBuilder<MainScreenCubit, MainScreenState>(
        builder: (context, state){
          if(state is MainScreenUpdateCounterState)
            return Center(child: Text('${state.value}', style:
Theme.of(context).textTheme.headline2));
          return Container();
        },
      ),
      floatingActionButton: Column(
        mainAxisAlignment: MainAxisAlignment.end,
        crossAxisAlignment: CrossAxisAlignment.end,
        children: <Widget>[
          FloatingActionButton(
            child: const Icon(Icons.add),
            onPressed: () =>
BlocProvider.of<MainScreenCubit>(context).addValue(),
          ),
          const SizedBox(height: 8),
          FloatingActionButton(
            child: const Icon(Icons.remove),
            onPressed: () =>
BlocProvider.of<MainScreenCubit>(context).removeValue(),
          ),
        ],
      ),
    );
  }
}
```

Рисунок 24 – Экран приложения

Чтобы кубит работал, весь экран необходимо обернуть в ***BlocProvider*** в файле main_screen_provider (рисунок 25).

```
class MainScreenProvider extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider<MainScreenCubit>(
      create: (context) => MainScreenCubit(),
      child: MainScreen(),
    );  // BlocProvider
  }
}
```

Рисунок 25 – Провайдер кубита

Таким образом получили полноценную систему управления состояниями экрана мобильного приложения на примере счетчика (рисунок 26).

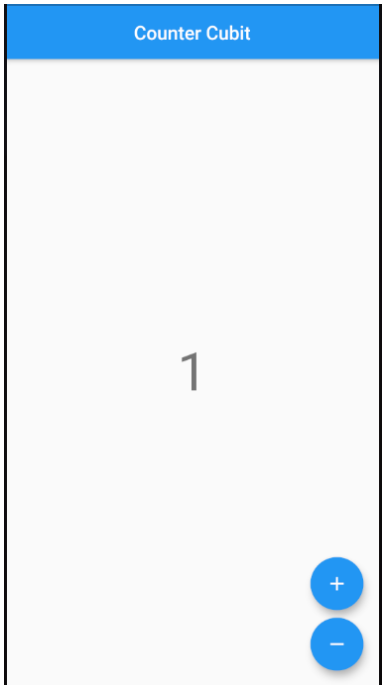


Рисунок 26 – Реализованный счетчик, основанный на кубите

Сам по себе кубит используется для более сложных экранов, но был рассмотрен на простом примере. Таким образом, произошло отделение уровня представления от уровня бизнес-логики. Также за счет различных состояний можно менять и полностью содержимое экрана.

Порядок выполнения работы

1. Изучить теоретические основы.
2. За основу данной лабораторной работы взять проект и задание из прошлой.
3. Реализовать кубит и состояние кубита.
4. Вынести все расчеты в кубит.
5. Вместо переключения экранов приложения реализовать смену состояний.
6. Подготовится к контрольным вопросам.

Контрольные вопросы

1. Виджет BlocProvider.
2. Виджет BlocBuilder.
3. Класс Cubit.
4. Вызов метода Cubit.
5. Функция emit.

1.5 Лабораторная работа «Хранение данных»

Цель работы

Ознакомиться с инструментами долговременного хранения информации и научиться использовать их.

Форма проведения

Выполнение индивидуального задания

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

SharedPreferences

SharedPreferences – постоянное хранилище, используемое приложениями для хранения простых данных. Это хранилище является относительно постоянным, пользователь может зайти в настройки приложения и очистить данные приложения, тем самым очистив все данные в хранилище. Принимает данные в формате ключ – значения. Поддерживаемые типы данных: int, double, bool, String и List<String>.

На рисунке 27 представлен процесс инициализации Shared и занесение данных разного типа.

```
final prefs = await SharedPreferences.getInstance();
await prefs.setInt('counter', 10);
await prefs.setBool('repeat', true);
await prefs.setDouble('decimal', 1.5);
await prefs.setString('action', 'Start');
await prefs.setStringList('items', <String>['Earth', 'Moon', 'Sun']);
```

Рисунок 27 – Занесение различных типов данных

Получение данных строится похожим образом, что и занесение. Но не стоит забывать про нулевую безопасность, в случае если под искомым ключом ничего не хранится – результатом будет null. Процесс чтения показан на рисунке 28.

```
final int? counter = prefs.getInt('counter');
final bool? repeat = prefs.getBool('repeat');
final double? decimal = prefs.getDouble('decimal');
final String? action = prefs.getString('action');
final List<String>? items = prefs.getStringList('items');
```

Рисунок 28 – Чтение различных типов данных

Для удаления записи используется метод remove (рисунок 29).

```
final success = await prefs.remove('counter');
```

Рисунок 29 – Удаление данных

SQflite

SQflite – библиотека для Flutter для работы с реляционными базами данных, которая совмещает в себе элементы ORM и SQL DDL. Наиболее надёжное средство хранения данных с поддержкой миграций.

Разберем на примере хранения личной информации работу с данной библиотекой. Для начала создадим класс ***DBProvider*** основываясь на паттерне «Одиночка» (рисунок 30). В момент обращения к базе данных срабатывает геттер database. Если переменная базы данных существует, он вернет ее, если нет, – вызовет функцию ***_initDB(),*** которую рассмотрим позже. Реализация данного паттерна необходима для того, чтобы экземпляр базы данных не пересоздавался при каждом обращении. За счет этого повышается скорость работы и безопасность хранимых данных.

```
class DBProvider {
  DBProvider._();

  static final DBProvider db = DBProvider._();

  static Database? _database;

  Future<Database> get database async {
    if (_database != null) {
      return _database!;
    }

    _database = await _initDB();
    return _database!;
  }
  Future<Database> _initDB() async {

  }
}
```

Рисунок 30 – Реализация класса DBProvider

Далее реализуем функцию _initDB() (рисунок 31). Данная функция инициализирует базу данных. Сначала функцией ***getApplicationDocumentsDirectory***() получаем внутреннюю папку приложения, где и будет храниться база данных. Добавляем к пути название базы данных и возвращаем вызов метода ***openDatabase(),*** принимающий путь, версию базы данных и функцию на случай если БД не существует. При изменении структуры базы данных необходимо изменить ее версию, тогда при запуске сработает функция создания базы данных и на старую структуру «накатится» миграция.

```
Future<Database> _initDB() async {
  Directory dir = await getApplicationDocumentsDirectory();
  String path = dir.path + 'mybase.db';
  return await openDatabase(path, version: 1, onCreate: _createDB);
}
```

Рисунок 31 – Метод инициализации

Далее реализуем функцию _createDB() (рисунок 32). Данная функция создает структуру базы данных. Функция имеет 2 обязательных аргумента Database и int, первый необходим для обращения в базу данных, второй номер версии. Для создания таблицы необходимо обратиться к переменной db и вызвать метод execute. В метод execute передается SQL код для создания таблицы.

```
Future<void> _createDB(Database db, int version) async {
  await db.execute("""CREATE TABLE Person(
                              id INTEGER PRIMARY KEY AUTOINCREMENT,
                              name TEXT,
                              phone TEXT,
                          );""");
}
```

Рисунок 32 – Метод создания структуры

Далее реализуем два простейших метода на добавление и чтение (рисунок 33). Для добавления необходимо использовать метод db.insert(), где в качестве параметров принимается название таблицы и Map<String, dynamic>. Для чтения данных используется метод db.query() принимающий только название таблицы и возвращающий список всех записей в формате Map<String, dynamic>.

```
Future<void> addPerson(Map<String, dynamic> person) async {
  Database db = await this.database;
  await db.insert("Person", person);
}
Future<List<Map<String,dynamic>>> getAllPerson() async {
  Database db = await this.database;
  final List<Map<String, dynamic>> data = await db.query("Person");
  return data;
}
```

Рисунок 33 – Методы записи и чтения

Для получения конкретной записи необходимо создать новый метод доработав getAllPerson (рисунок 34). Для поиска конкретной записи мы в методе db.query передаем дополнительные параметры where и whereArgs. Where указывает по какому полю мы будет делать запрос, а whereArgs по каким значениям. В примере реализован запрос по ID.

```
Future<List<Map<String,dynamic>>> getPerson(int id) async {
  Database db = await this.database;
  final List<Map<String, dynamic>> data;
  data = await db.query("Person", where: "id = ?", whereArgs: [id]);
  return data;
}
```

Рисунок 34 – Метод чтения конкретной записи

Далее реализуем метод удаления конкретной записи (рисунок 35),

```
Future<int> deletePerson(int id) async {
  Database db = await this.database;
  return await db.delete("Person", where: "id = ?", whereArgs: [id]);
```

Рисунок 35 – Удаление конкретной записи

Также нам необходимо создать метод редактирования данных (рисунок 36). Данный метод принимает новые данные и id и на основе их заменят обновившиеся значения.

```
Future<int> updatePerson(Map<String, dynamic> person, int id) async {
  Database db = await this.database;
  return await db.update("Person", person, where: "id = ?", whereArgs: [id]);
}
```

Рисунок 36 – Обновление конкретной записи

Таким образом реализовав все методы, получим класс, который полностью обеспечивает CRUD. Работа с базами данных в лучшем виде раскрывается при работе с моделями, но для получения базовых навыков и решения не трудных задач достаточно работы с Map<String, dynamic>.

На рисунке 37 представлено взаимодействие с классом DBProvider из основного класса приложения.

```
DBProvider.db.addPerson({'name': "Roman", "phone": "88005553535"});
DBProvider.db.getAllPerson();
DBProvider.db.getPerson(1);
DBProvider.db.updatePerson({'name': "Roman Sergeevich", "phone": "3564"}, 1);
DBProvider.db.deletePerson(1);
```

Рисунок 37 – Взаимодействие с DBProvider

Порядок выполнения работы

1. Изучить теоретические основы.
2. За основу данной лабораторной работы взять проект и задание из предыдущей работы.
3. Реализовать сохранение всех вводимых данных и результатов расчетов.
4. Реализовать в кубите метод записи.
5. Добавить новый экран, где будут выводится списком вводимые ранее данные и результаты расчетов.
6. Для нового экрана реализовать кубит обеспечивающий загрузку данных.
7. На первом экране в leading AppBar добавить IconButton, и при нажатии на нее используя класс Navigator открыть новый экран с результатами.
8. Подготовится к контрольным вопросам.

Варианты заданий

1. Реализовать с использованием SQflite.
2. Реализовать с использованием SharedPreferences.
3. Реализовать с использованием GetStorage.
4. Реализовать с использованием SQflite.
5. Реализовать с использованием SharedPreferences.
6. Реализовать с использованием GetStorage.
7. Реализовать с использованием SQflite.
8. Реализовать с использованием SharedPreferences.
9. Реализовать с использованием GetStorage.
10. Реализовать с использованием SQflite.

Контрольные вопросы

1. Взаимодействие с SQflite.
2. Взаимодействие с SharedPreferences.

1.6 Лабораторная работа «Взаимодействие с сетью»

Цель работы

Научиться взаимодействовать с API посредствам http-запросов из мобильного приложения.

Форма проведения

Выполнение индивидуального задания

Форма отчетности

На проверку должен быть предоставлен исходный код мобильного приложения, разработанного в рамках индивидуального задания.

Теоретические основы

Для взаимодействия с сетью во Flutter предусмотрено множество инструментов, основным является HTTP-запросы. Для работы с HTTP-запросами предусмотрена библиотека http, которую можно найти на pub.dev. Данная библиотека реализует весь необходимый функционал, для работы с запросами и не только.

На примере рассмотрим взаимодействие мобильного приложения с REST API. Рассматриваться будет на основе открытого API NASA.

Для начало необходимо перейти на сайт https://api.nasa.gov для получения ключа для работы с API или можно использовать уже сгенерированный eQnprvXukgfNomTanZiHT1DqLApcABzFjI350dyZ.

После получения ключа в обзоре всех доступных API найдем Mars Rover Photos и найдем примеры запросов (рисунок 38).

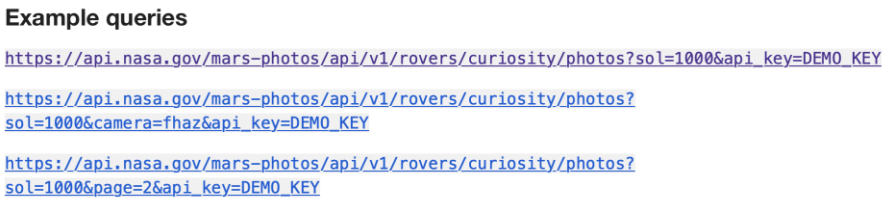


Рисунок 38 – Примеры запросов.

Для начала изучим запрос. Основное, с чем придется работать – это название марсохода, сол и ключ API. Название марсохода это часть пути, а вот сол и ключ API уже параметры. Эту информацию необходимо будет учитывать для построения HTTP-запроса в мобильном приложении.

Далее необходимо сделать запрос по этой ссылке что бы изучить структуру ответа. Для этого достаточно внести эту ссылку в строку ссылки браузера или просто нажать на нее. Полученный результат будет являться неструктурированным и плохо читаемым JSON форматом (рисунок 39). Данный формат является основным при работе с API.

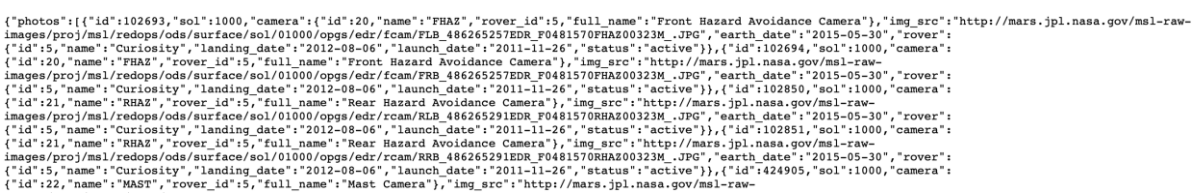


Рисунок 39 – Структура ответа в формате JSON

Для взаимодействия приложения с большими объектами из сети часто используют модели. Модель в нашем случае – это представление JSON объекта в виде класса на языке Dart. Ниже представлен простой JSON объект в виде объекта Dart (рисунок 40).

Рисунок 40 – Представление JSON в виде Dart класса

Данная модель представляет собой типы данных и название полей сформированных на основе JSON объекта и два метода. Первый метод fromJson() необходим для создания экземпляра класса на основе проходимого от сервера ответа в виде JSON. Второй метод toJson() необходим для обратного процесса – когда нам необходимо создать JSON объект на основе модели, например, для отправки на сервер.

Существует множество методов создания моделей: ручное написание, автогенерация и использование специальных инструментов для формирования моделей. В текущей ситуации следует рассматривать именно специальные инструменты, так как ручное создание для больших моделей – процесс трудоемкий, и можно допустить множество ошибок, а автогенерация не всегда надежна и местами трудно выстроить процесс генерации.

После того как ознакомились с моделями перейдем на сайт https://javiercbk.github.io/json_to_dart/ (Сервис трансляции JSON в Dart) и вставим наш ответ от сервера в поле JSON, зададим имя классу и нажмем кнопку Generate Dart. В результате получим Dart код, который в дальнейшем можно использовать в проекте (рисунок 41).

Рисунок 41 – Результат трансляции JSON в Dart

В результате появилось 4 класса, но при этом JSON была одна. Получилось это из-за того, что одни объекты являются частями других объектов и находятся в них (часто в виде списков). На рисунке 42 приведен пример того, как объект «Студен» является вложенным относительно объекта «Факультет» поэтому в Dart коде существует 2 класса.

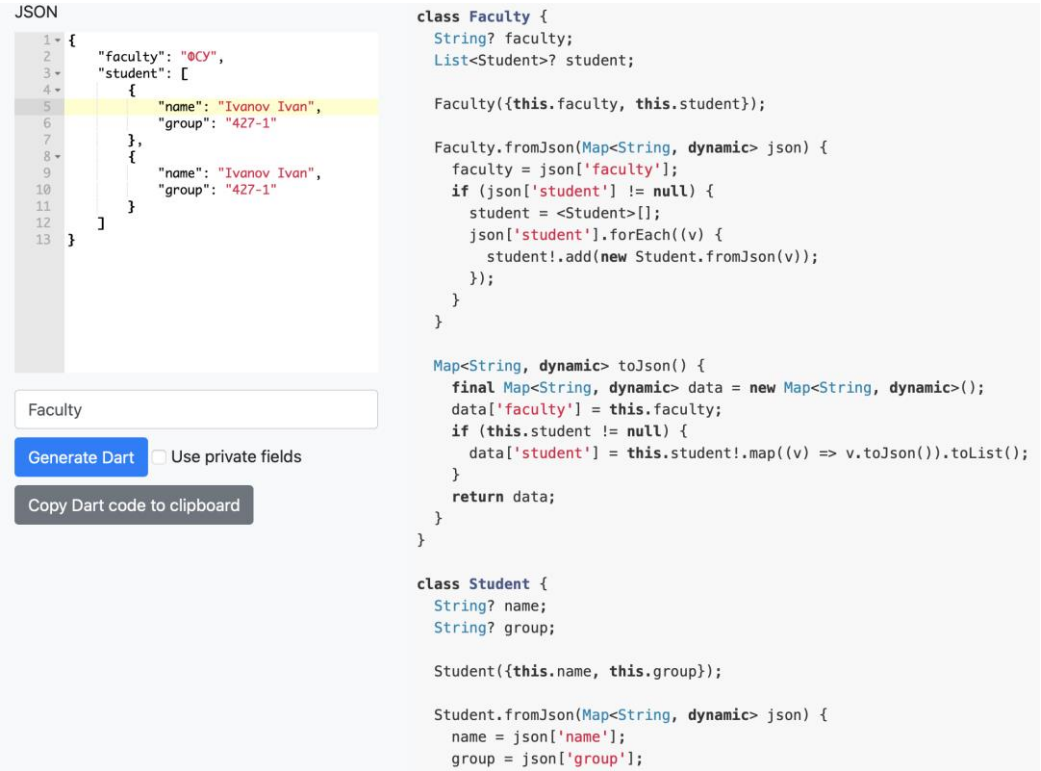


Рисунок 42 – Вложенные модели

На рисунке 43 схематична изображена структура вложенности моделей. Класс Photos содержит в себе ссылку на фотографию, модель марсохода (Rover), модель камеры (Camera) и прочую информацию. Модели Rover и Camera содержат информацию о марсоходе и камере соответственно.

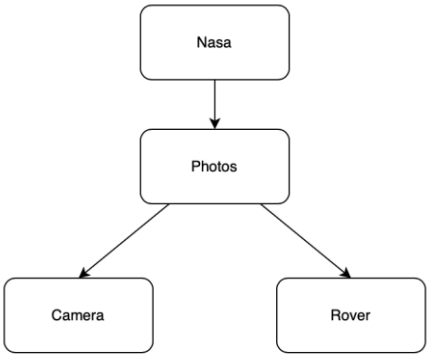


Рисунок 43 – Схема вложенности моделей

Далее необходимо перенести сгенерированные модели в проект. В папке lib создадим папку models и далее для каждого класса необходимо создать свой файл (рисунок 44).

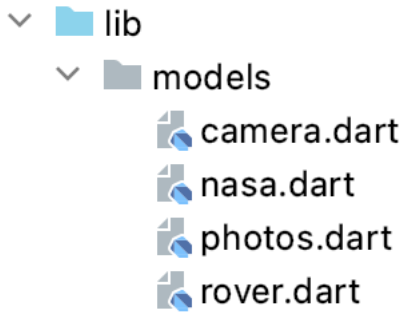


Рисунок 44 – Структура проекта для хранения классов моделей

После подготовки моделей, необходимо написать класс запросов, использующих библиотеку http и добавить библиотеку в зависимости. В папке lib создадим папку requests, где и будут храниться классы взаимодействия с сервером. Создадим файл api.dart и функцию запроса (рисунок 45).

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

Future<Map<String, dynamic>> getNasaData() async {
  Uri url = Uri.parse('https://api.nasa.gov/mars-photos/api/v1/rov
  final response = await http.get(url,);

  if(response.statusCode == 200) {
    return json.decode(response.body);
  } else {
    throw Exception('Error: ${response.reasonPhrase}');
  }
}
```

Рисунок 45 – Функция запроса

В данной функции создаем URI, исходя из ссылки, с помощью метода Uri.parse(). Далее с помощью метода http.get() выполняется HTTP GET запрос, в качестве аргумента передаем Uri. После получения ответа необходимо убедится в том, что, ответ пришел с успешным кодом. Если посмотреть на коды ответов HTTP – успешным является код «200», получить код можем, обратившись к полю statusCode ответа. Если код успешный, то возвращаем данные, в противном случае выбрасываем ошибку.

После создания функции запроса необходимо создать кубит и его состояния. Будет 3 состояния экрана: состояние пока идет загрузка, когда загружено и когда ошибка. Состояние, когда загружено будет содержать модели, которые мы создадим в самом кубите. Состояния представлены на рисунке 46.

```dart
abstract class NasaState{}

class NasaLoadingState extends NasaState{}

class NasaLoadedState extends NasaState{
  Nasa data;
  NasaLoadedState({required this.data});
}

class NasaErrorState extends NasaState{}
```

Рисунок 46 – Состояния кубита

Далее создадим кубит (рисунок 47). В данном купите стартовое состояние будет являться состояние загрузки. В методе loadData содержится обработчик ошибок. В случае удачного выполнения будет «запущено» состояние, когда данные загружены. В случае возникновения ошибки, будет «запущено» состояние ошибки.

```dart
class NasaCubit extends Cubit<NasaState>{
  NasaCubit() : super(NasaLoadingState());

  Future<void> loadData()async {
    try{
      Map<String, dynamic> apiData = await getNasaData();
      Nasa nasaData = Nasa.fromJson(apiData);
      emit(NasaLoadedState(data: nasaData));
      return;
    }catch(e){
      emit(NasaErrorState());
      return;
    }

  }
}
```

Рисунок 47 – Кубит с обработкой ошибок

Далее создадим BlocProvider и BlocBuilder кубита. В BlocBuilder обрабатывается отображение при трех состояниях. Стартовым является NasaLoadingState. В этом состоянии мы вызываем функцию кубита, а сами отображаем пользователю индикатор загрузки (рисунок 48).

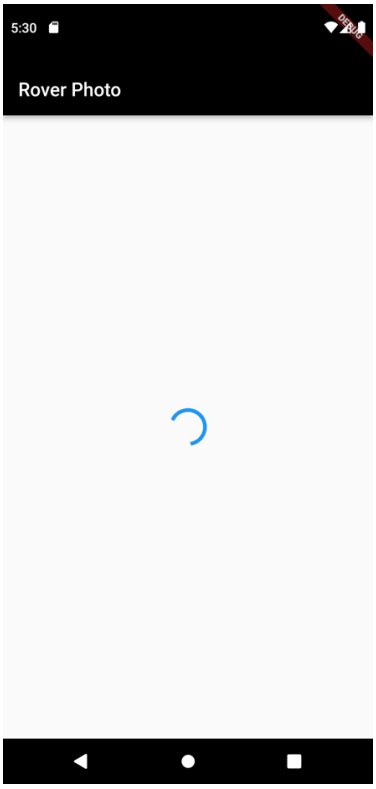


Рисунок 48 – Отображение при состоянии загрузки

При возникновении ошибки во время загрузки данных сработает состояние NasaErrorState. Вызов данного состояния приведет к перезагрузке BlocBuilder и «заходу» в новое состояние. Экран состояния ошибки изображен на рисунке 49.

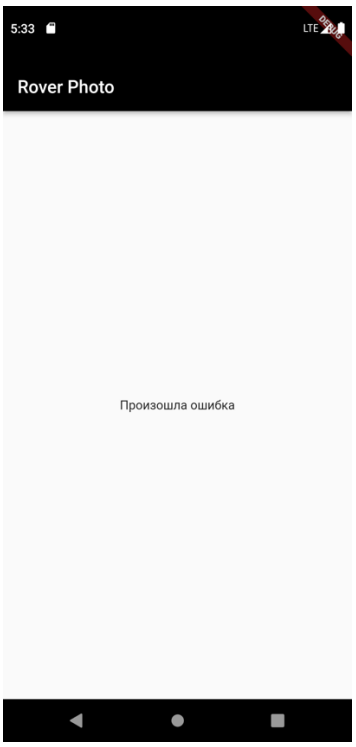


Рисунок 49 – Отображение при состоянии ошибки

Если во время загрузки ошибка не произошла, вызывается состояние NasaLoadedState и передается в него модели отображаемые на экране (рисунок 50).

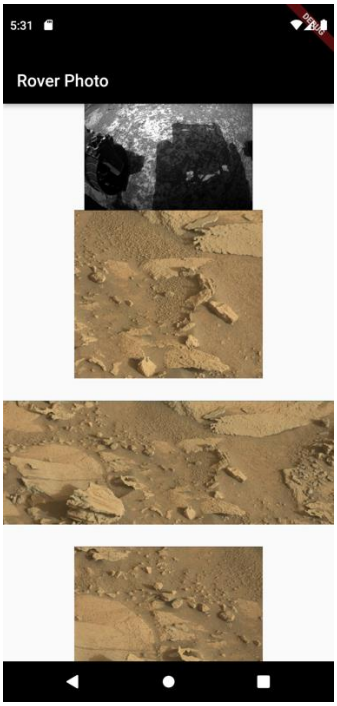


Рисунок 50 – Отображение при состоянии успешной загрузки

Полный код BlocBuilder представлен на Рисунке 51.

```
return BlocBuilder<NasaCubit, NasaState>(builder: (context, state){
  if (state is NasaLoadingState) {
    BlocProvider.of<NasaCubit>(context).loadData();
    return const Center(child: CircularProgressIndicator());
  }
  if (state is NasaLoadedState) {
    return ListView.builder(
      itemCount: state.data.photos!.length,
      itemBuilder: (context, index) {
        return Container(
          height: 200,
          width: 200,
          child: Image.network(state.data.photos![index].imgSrc!),
        );  // Container
      },
    );  // ListView.builder
  }
  if (state is NasaErrorState) {
    return const Center(child: Text("Произошла ошибка"));
  }
```

Рисунок 51 – Реализация BlocBuilder.

Из-за большего объема данных, получаемых с сети, рекомендуется использовать ListView вместо Column и Row, так как ListView переиспользует выделяемую память, а другие

виджеты компоновки нет. В приведенном примере использование Column из-за картинок привело бы к полной загрузке ОЗУ, а следом и отключению приложения на уровне ОС.

Приведенный пример показал взаимодействия мобильного приложения с REST API посредствам HTTP GET запроса и в полной мере показал использование кубита.

Порядок выполнения работы

1. Изучить теоретические основы.
2. За основу взять приведенные в теоретическом материале примеры.
3. Настроить запрос в зависимости от своего варианта.
4. Проанализировать приходящие данные.
5. Разработать мобильное приложение отражающее максимальное количество данных с API и при этом сохраняющее эргономику, красоту и удобство использования.
6. Подготовится к контрольным вопросам.

Варианты заданий

1. Марсоход – curiosity, сол (марсианский день) – 1000, открытие нового экрана на основе Navigator.
2. Марсоход – curiosity, сол (марсианский день) – 100, изменение состояния на основе Cubit.
3. Марсоход – opportunity, сол (марсианский день) – 100, изменение состояния на основе Cubit.
4. Марсоход – spirit, сол (марсианский день) – 100, открытие нового экрана на основе Navigator.
5. Марсоход – opportunity, сол (марсианский день) – 150, изменение состояния на основе Cubit.
6. Марсоход – curiosity, сол (марсианский день) – 50, открытие нового экрана на основе Navigator.
7. Марсоход – spirit, сол (марсианский день) – 50, изменение состояния на основе Cubit.
8. Марсоход – opportunity, сол (марсианский день) – 130, изменение состояния на основе Cubit.
9. Марсоход – spirit, сол (марсианский день) – 90, открытие нового экрана на основе Navigator.
10. Марсоход – opportunity, сол (марсианский день) – 110, изменение состояния на основе Cubit.

Контрольные вопросы

1. Вложенные модели.
2. Библиотека http.
3. Виды HTTP запросов.
4. Использование виджетов компановки в задачах взаимодействия с сетью.
5. Обработка ошибок.
6. Методы создания моделей.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

2.1 Общие положения

Целями самостоятельной работы являются систематизация, расширение и закрепление теоретических знаний в области разработки мобильных приложений.

Самостоятельная работа студента по дисциплине «Разработка мобильных приложений» включает следующие виды деятельности:

1) проработка лекционного материала, в том числе подготовка к тестированию;

2) подготовка к лабораторным работам;

3) подготовка к промежуточной аттестации.

В ходе самостоятельной работы студент, ориентируясь на изложенные рекомендации, планирует свое время и перечень необходимых работ в зависимости от индивидуальных психофизических особенностей. Формат самостоятельной работы студентов может отличаться в зависимости от формы обучения и объема аудиторной работы.

2.2 Проработка лекционного материала и подготовка к лабораторным работам

Для качественного усвоения учебного материала целесообразно осуществлять проработку лекционного материала, которая направлена как на систематизацию имеющегося материала, так и на подготовку к освоению практических аспектов, связанных с содержанием дисциплины.

Проработка лекционного материала включает деятельность, связанную с изучением рекомендуемых преподавателем источников, в которых отражены основные моменты, затрагиваемые в ходе лекций. Кроме того, важное место отведено работе с собственноручно составленным конспектом лекций. При конспектировании во время лекции помните, что не следует записывать все, что говорит и/или демонстрирует лектор: старайтесь выявить главное и записать только это. Цель конспекта – формирование целостного логически выстроенного взгляда на круг вопросов, затрагиваемых в ходе изучения соответствующей темы, а не механическая фиксация текстовой и графической информации.

Во внеаудиторное время проработка лекционного материала может быть выстроена в двух основных форматах:

а) отработка прослушанной лекции (прочтение конспекта и рекомендованных преподавателем источников с сопоставлением записей) и восполнение пробелов, если они имелись (например, если студент не понял чего-то, не успел записать);

б) прочтение перед каждой последующей лекцией предыдущей, дабы не тратилось много времени на восстановление контекста изучения дисциплины при продолжающейся или связанной теме.

В ходе проработки лекционного материала обращайте внимание на контрольные вопросы, которые, как правило, имеются в конце каждой темы учебника (учебного пособия). Отвечая на них, можно сделать вывод о степени понимания материала. Если ответы на какие-то вопросы вызвали затруднения, то следует предпринять еще одну попытку изучения отдельных вопросов.

При подготовке к лабораторным работам необходимо заранее изучить методические рекомендации по его проведению, обратить внимание на цель, формат и содержание занятия.

Если какие-то моменты вызвали дополнительные вопросы, целесообразно обратиться к содержанию лекционного материала, рекомендациям преподавателя по изучению теоретической части курса (рекомендуемым источникам) или за личной консультацией. В ходе подготовки к лабораторным работам может потребоваться обращение к различным источникам. Проявляйте инициативу и самостоятельность в данном вопросе. При этом следует пользоваться только авторитетными изданиями, как печатными, так и электронными.

2.3 Подготовка к промежуточной аттестации

Подготовка к экзамену (зачету) включает в себя изучение теоретического материала, представляющего в интегративном виде содержание дисциплины. Экзаменационный (зачетный) билет содержит по 2 теоретических вопроса.

СПИСОК ЛИТЕРАТУРЫ

1. Соколова, В. В. Вычислительная техника и информационные технологии. Разработка мобильных приложений : учебное пособие для вузов / В. В. Соколова. – Москва : Издательство Юрайт, 2018. – 175с. – URL:https://urait.ru/bcode/414105.