МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«МИРЭА – Российский технологический университет»**
**РТУ МИРЭА**

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

# Отчет по практической работе №6

по дисциплине «Разработка кроссплатформенных мобильных приложений»

**Выполнил:**
Студент группы ИКБО-07-22 Гутиева В.А.

**Проверил:**
Старший преподаватель кафедры МОСИТ Шешуков Л.С.

Москва 2025 г.

# 1. Подключение внешних зависимостей во Flutter

В разработке приложений на Flutter часто возникает необходимость в использовании внешних библиотек (пакетов), которые расширяют функциональность базового фреймворка. Для управления этими зависимостями используется файл pubspec.yaml, который является конфигурационным файлом проекта. В нём определяются два ключевых раздела для зависимостей:

- **dependencies**: Основные зависимости, необходимые для работы приложения в продакшене (например, библиотеки для сетевых запросов, UI-компонентов или интеграции с сервисами).
- **dev_dependencies**: Зависимости для разработки и тестирования (например, инструменты для unit-тестов или линтинга кода), которые не включаются в финальную сборку приложения.

Процесс добавления зависимости включает указание её имени, источника и, при необходимости, версии. После редактирования pubspec.yaml выполняется команда flutter pub get (или автоматически в IDE), чтобы скачать и установить пакеты. Зависимости классифицируются по источникам на четыре типа с точки зрения источников.

## 1.1. Зависимости из облаков

Pub.dev (https://pub.dev) — это официальный публичный репозиторий пакетов для языков Dart и фреймворка Flutter, поддерживаемый Google и сообщество (Рисунок 1). Он содержит тысячи пакетов, включая как открытые от разработчиков, так и официальные от команды Flutter.

Рисунок 1 – Сайт pub.dev

Страница каждого пакета содержит его название, актуальную версию, дату публикации, автора и другую информацию, которая может быть полезна при использовании пакета.

Для подключения зависимости в разделе dependencies pubspec.yaml требуется указать имя пакета и версию (если требуется).

Подключим пакет firebase_messaging версии 14.6.1 отмеченный как Flutter Favorite, что указывает на его стабильность и рекомендованность для интеграции push-уведомлений от Firebase (Рисунок 2).

```
dependencies:
  firebase_messaging: ^14.6.1
```

Рисунок 2 – Подключение зависимости из pub.dev

## 1.2. Зависимости из Git

Не все пакеты публикуются на pub.dev — иногда они находятся в приватных или открытых репозиториях на платформах вроде GitHub, GitLab или Bitbucket. Это полезно для форков, экспериментальных версий или внутренних библиотек компании.

При подключении зависимости из Git-а требуется указать в обязательном порядке маршрут до пакета, а также ветку или коммит, который требуется подключить. Полный синтаксис подключения пакета из Git-а можно увидеть на рисунке 4.

Рисунок 4 –Подключения зависимости из Git

### 1.3. Локальные зависимости

Для небольших, внутренних пакетов, которые не стоит выносить в отдельный репозиторий, используется локальное размещение. Такие пакеты всё равно являются полноценными Dart-пакетами, но хранятся в поддиректории проекта.

Для подключения нужно прописать полный или относительный путь до этой зависимости. Пример подключения локальной зависимости изображен на рисунке 5.

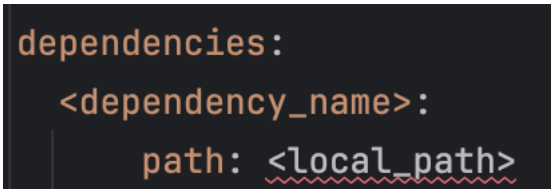

Рисунок 5 – Подключение локальной зависимости

## 2. Версионирование зависимостей

Версионирование — ключевой аспект управления зависимостями, чтобы обеспечить совместимость и стабильность. В Dart/Flutter используется семантическое версионирование (SemVer), где версия представлена в формате major.minor.patch (например, 1.2.3):

- **Major (мажор)**: увеличивается при breaking changes — изменениях API, которые ломают обратную совместимость. Это может потребовать переписывания кода в вашем приложении.
- **Minor (минор)**: увеличивается при добавлении новой функциональности без нарушения совместимости. Код, работавший с предыдущей версией, продолжит работать.

- **Patch (патч)**: увеличивается при исправлении багов, не затрагивающих API. Это чистые фиксы для стабильности.

В pubspec.yaml версии указываются с использованием constraints (ограничений) — операторов, которые определяют диапазон допустимых версий. Pub (менеджер пакетов Dart) автоматически выберет наилучшую версию в пределах ограничений, учитывая зависимости других пакетов.

Основные операторы:

- any: любая версия (эквивалентно отсутствию указания версии). Полезно для быстрого прототипирования, но рискованно для продакшена из-за возможных breaking changes.
- x.y.z: точная версия (например, 1.2.3). Фиксирует на конкретной, избегая обновлений.
- >x.y.z: строго выше указанной (например, >1.2.3 — версии от 1.2.4 и выше).
- >=x.y.z: выше или равна (например, >=1.2.3 — от 1.2.3 и выше).
- <x.y.z: строго ниже (например, <2.0.0 — версии до 1.99.99).
- <=x.y.z: ниже или равна (например, <=1.2.3 — до 1.2.3 включительно).
- ^x.y.z (caret): Диапазон для совместимых обновлений. Для major > 0: >=x.y.z < (x+1).0.0 (например, ^1.2.3 — от 1.2.3 до 1.99.99, но не 2.0.0). Для major = 0 (бета-версии): >=0.y.z < 0.(y+1).0 (например, ^0.1.2 — от 0.1.2 до 0.1.99, но не 0.2.0), так как минорные изменения могут быть breaking.

## 3. Кеширование изображений из сети

В приложениях, работающих с сетевыми изображениями (например, соцсети, галереи), повторная загрузка одного и того же изображения приводит к неэффективному использованию трафика, нагрузке на сервер и задержкам для пользователя. Пример: в ленте новостей пользователь видит миниатюру поста, затем открывает детали и добавляет в избранное — без кэша

изображение загружается заново каждый раз, что ухудшает UX.

Flutter имеет встроенный механизм, позволяющий управлять файлами в определенном каталоге, предназначенном для временных файлов. Основная цель этого каталога — хранить временные данные, которые могут быть восстановлены приложением в случае их удаления. Цель этого каталога — улучшить общую производительность или каким-либо образом улучшить взаимодействие с пользователем. Тем не менее, управление этими файлами может быть скучным и утомительным. Чтобы упростить этот процесс, у нас есть пакет cached_network_image, который абстрагирует нас от управления кэш-памятью. Этот пакет позаботится о загрузке изображения в первый раз, сохранит его в кеше и извлечет его оттуда, если то же изображение будет запрошено снова.

## 3.1. Подключение в проект

Для подключения пакета cached_network_image версии 3.4.1 в проект требуется в файл pubspec.yaml добавить пакет в dependency. Пример добавления изображен на рисунке 6. Данная версия выбрана так как является самой последней и актуальной.


Рисунок 6 – Добавление пакета cached_network_image в приложение

Альтернативным вариантом является введение команды «flutter pub add cached_network_image» в терминале, смотрящим в директорию проекта (Рисунок 7).

```
→  pr2 git:(pr6) flutter pub add cached_network_image
Resolving dependencies...
Downloading packages... (1.6s)
+ cached_network_image 3.4.1
+ cached_network_image_platform_interface 4.1.1
+ cached_network_image_web 1.3.1
  characters 1.4.0 (1.4.1 available)
+ crypto 3.0.6
+ ffi 2.1.4
+ file 7.0.1
+ fixnum 1.1.1
+ flutter_cache_manager 3.4.1
  flutter_lints 5.0.0 (6.0.0 available)
+ http 1.5.0
+ http_parser 4.1.2
  lints 5.1.1 (6.0.0 available)
  material_color_utilities 0.11.1 (0.13.0 available)
  meta 1.16.0 (1.17.0 available)
+ octo_image 2.1.0
+ path_provider 2.1.5
+ path_provider_android 2.2.20
+ path_provider_foundation 2.4.3
+ path_provider_linux 2.2.1
+ path_provider_platform_interface 2.1.2
+ path_provider_windows 2.3.0
+ platform 3.1.6
+ plugin_platform_interface 2.1.8
+ rxdart 0.28.0
+ sprintf 7.0.0
+ sqflite 2.4.2
+ sqflite_android 2.4.2+2
+ sqflite_common 2.5.6
+ sqflite_darwin 2.4.2
+ sqflite_platform_interface 2.4.0
+ synchronized 3.4.0
  test_api 0.7.6 (0.7.7 available)
+ typed_data 1.4.0
+ uuid 4.5.1
+ web 1.1.1
+ xdg_directories 1.1.0
Changed 31 dependencies!
6 packages have newer versions incompatible with dependency constraints.
```

Рисунок 7 – Добавление пакета альтернативным вариантом

## 3.2. Отображение картинки

Пакет cached_network_image предоставляет доступ к Widget-у CachedNetworkImage. Данный Widget создан для отображения изображения из сети интернет с последующим его кешированием. Для установки искомого изображения необходимо передать сетевой маршрут до него в аргумент Widget-а imageUrl. При первом CachedNetworkImage запросе на рендеринг этого изображения он загрузит его из Интернета и сохранит, а затем будет извлекать его из кеша. Пример использования CachedNetworkImage изображен на рисунке 8.

```
CachedNetworkImage(
  imageUrl: _url,
  progressIndicatorBuilder: (context, url, progress) =>
  const CircularProgressIndicator(),
  errorWidget: (context, url, error) => const Center(
    child: Icon(
      Icons.error,
      color: Colors.red,
    ), Icon
  ), Center
), CachedNetworkImage
```

Рисунок 8 – Пример использования CachedNetworkImage

# 4. Выполнение практической работы №6

Для обеспечения корректной работы приложения добавим пакет cached_network_image. Добавления зависимости проиллюстрирован на рисунке 9.

```
dependencies:
  flutter:
    sdk: flutter
  cached_network_image: ^3.4.1
```

Рисунок 9 – Добавление разрешения для доступа в интернет

В рамках модернизации приложения для трекера задачек, разработанного в предыдущих практических работах, я добавила поддержку сетевых изображений (обложек задач).

Теперь каждая таска отображается с собственной обложкой, загружаемой из интернета с использованием пакета cached_network_image. Это позволяет не только улучшить визуальное восприятие приложения, но и обеспечивает кэширование изображений — после первой загрузки они отображаются даже без подключения к сети.

Загрузка обложек реализована с помощью виджета CachedNetworkImage, который автоматически сохраняет изображения в локальный кэш. В момент загрузки отображается индикатор прогресса, а при

ошибке — иконка повреждённого изображения. После первой загрузки обложки становятся доступны в офлайн-режиме, что подтверждается тестированием при отключённой сети.

Каждая таска оформлена с обложкой (размером 60x60), названием, описанием, статусом, возможностью смены статуса и кнопкой удаления. Таким образом, приложение стало более наглядным и функциональным, сохранив при этом возможность офлайн-доступа к визуальным данным.

Реализация доработанного интерфейса с сетевыми изображениями представлена на рисунках 10–12.

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

class TaskCreateScreen extends StatefulWidget {
  @override
  _TaskCreateScreenState createState() => _TaskCreateScreenState();
}

class _TaskCreateScreenState extends State<TaskCreateScreen> {
  final _formKey = GlobalKey<FormState>();
  String _title = '';
  String _description = '';
  TaskStatus _status = TaskStatus.newTask;
  String? imageUrl;

  void _submit() {
    if (_formKey.currentState!.validate()) {
      _formKey.currentState!.save();
      final newTask = Task(
        title: _title,
        description: _description,
        status: _status,
        imageUrl: imageUrl,
      );
      Navigator.pop(context, newTask);
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Добавить задачу')),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              TextFormField(
                decoration: InputDecoration(labelText: 'Название'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Введите название задачи';
                  }
                  return null;
                },
                onSaved: (value) {
                  _title = value ?? '';
                },
              ),
              SizedBox(height: 15),


              TextFormField(
                decoration: InputDecoration(labelText: 'Описание'),
                maxLines: 3,
                onSaved: (value) {
                  _description = value ?? '';
                },
              ),
              SizedBox(height: 15),

              //текст для +картинки
              TextFormField(
                decoration: InputDecoration(labelText: "Ссылка на изображение"),
                onSaved: (value) => imageUrl = value,
              ),
              SizedBox(height: 15),

              DropdownButtonFormField<TaskStatus>(
                value: _status,
                decoration: InputDecoration(labelText: 'Статус'),
                items: TaskStatus.values.map((status) {
                  return DropdownMenuItem(
                    value: status,
                    child: Text(statusToString(status)),
                  );
                }).toList(),
                onChanged: (value) {
                  setState(() {
                    _status = value ?? TaskStatus.newTask;
                  });
                },
              ),
              SizedBox(height: 20),
              Center(
                child: ElevatedButton(
                  onPressed: _submit,
                  child: Text('Создать'),
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Рисунок 10 – Переделанный виджет

Для проверки функциональности кэширования сетевых изображений в приложении было проведено тестирование отображения пяти иконок на экране настроек. Тестирование включало два сценария: с активным подключением к интернету (для первоначальной загрузки и кэширования изображений) и без него (для проверки извлечения из локального кэша).

В первом случае изображения загружались с удаленных серверов и сохранялись в кэше с помощью пакета cached_network_image, обеспечивая плавный рендеринг (Рисунок 13-14).

Во втором случае, после отключения сети, приложение успешно отображало ранее закешированные изображения без задержек или ошибок, подтверждая оффлайн-доступность (Рисунок 15-16).

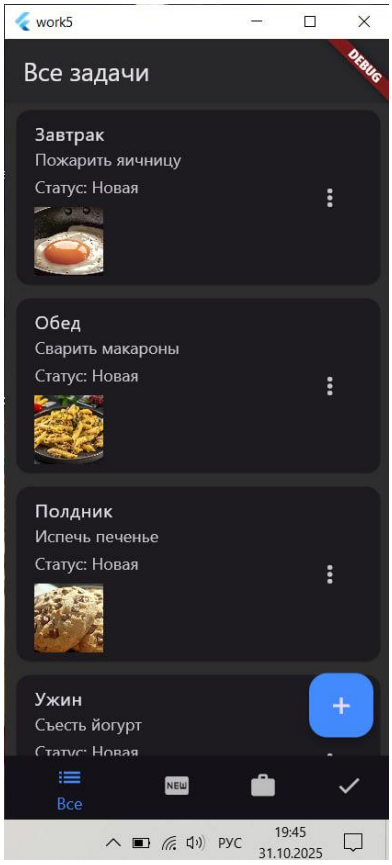


Рисунок 13 – Отображение 2 сетевых картинок с подключением к интернету

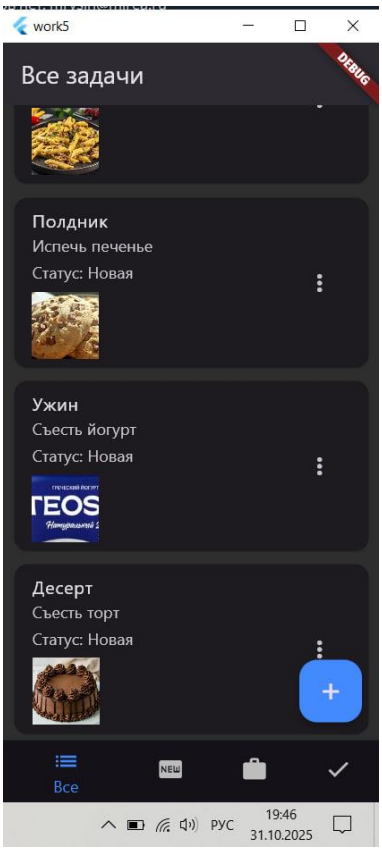


Рисунок 14 – Отображение 2 сетевых картинок с подключением к интернету

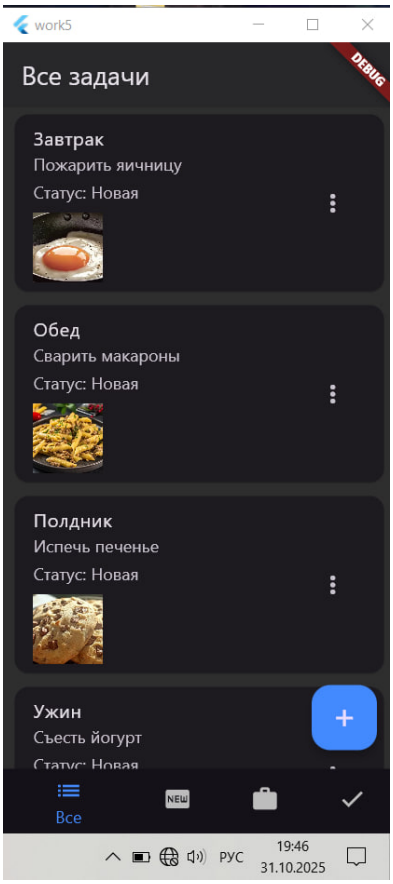


Рисунок 15 – Отображение 2 сетевых картинок без подключения к интернету

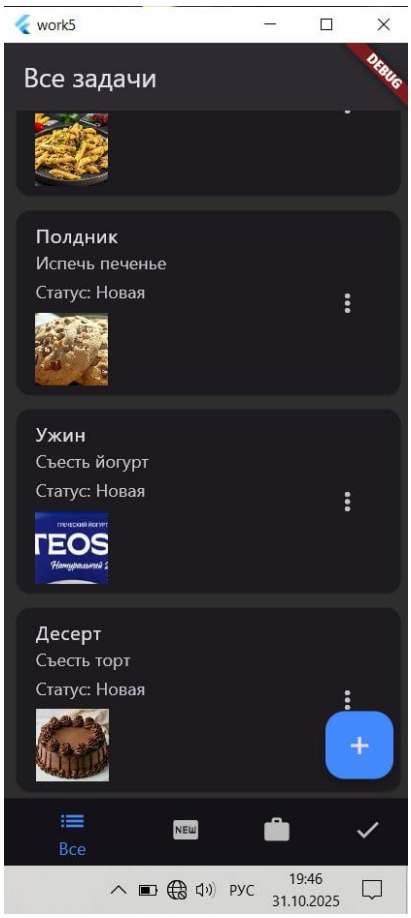


Рисунок 15 – Отображение 2 сетевых картинок без подключения к интернету

## Заключение

В ходе работы было модернизировано приложение для заметок из пятой практической работы путем добавления экрана настроек с пятью сетевыми изображениями, загружаемыми с использованием пакета `cached_network_image`. Это обеспечило кэширование для оффлайн-доступа. Тестирование подтвердило корректное отображение изображений как при наличии, так и при отсутствии сети.

Все изменения, выполненные в результате данной практической работы, были сохранены в удаленном репозитории github: https://github.com/viiLkaa/flutterWork6