



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

**Отчёт по практической работе №6**  
по дисциплине «Разработка кроссплатформенных мобильных приложений»

**Выполнили:**

Студенты группы ИКБО-07-22

Буренин А. А.

**Принял:**

Старший преподаватель кафедры  
МОСИТ

Шешуков Л.С.

Москва 2025

## 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 три основных типа.

### Зависимости из облаков

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

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). Например, для подключения пакета `cached_network_image` версии 3.4.1:

`dependencies:`

`cached_network_image: ^3.4.1`

### 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 требуется указать в обязательном порядке маршрут до пакета, а также ветку или коммит, который требуется подключить. Полный синтаксис подключения пакета из Git:

```
dependencies:  
  <dependency_name>:  
    git:  
      url: https://github.com/<repo_url>  
      ref: some-branch  
      path: some-path
```

## 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:

```
dependencies:  
  <dependency_name>:  
    path: <local_path>
```

## 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`:  $\geq x.y.z < (x+1).0.0$  (например, ^1.2.3 — от 1.2.3 до 1.99.99, но не 2.0.0). Для `major = 0` (бета-версии):  $\geq 0.y.z < 0.(y+1).0$  (например, ^0.1.2 — от 0.1.2 до 0.1.99, но не 0.2.0), так как минорные изменения могут быть *breaking*.

## 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 кеша 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, существует пакет `cached_network_image`, который абстрагирует от управления кеш-памятью. Этот пакет позаботится о загрузке изображения в первый раз, сохранит его в кеше и извлечёт его оттуда, если то же изображение будет запрошено снова.

## Подключение в проект

Для подключения пакета `cached_network_image` версии 3.4.1 в проект требуется в файл `pubspec.yaml` добавить пакет в `dependencies`. Данная версия выбрана так как является самой последней и актуальной на момент выполнения работы.

Альтернативным вариантом является введение команды «`flutter pub add cached_network_image`» в терминале, смотрящим в директорию проекта.

## Отображение картинки

Пакет `cached_network_image` предоставляет доступ к виджету `CachedNetworkImage`. Данный виджет создан для отображения изображения из сети интернет с последующим его кешированием. Для установки искомого изображения необходимо передать сетевой маршрут до него в аргумент виджета `imageUrl`. При первом запросе на рендеринг этого изображения `CachedNetworkImage` загрузит его из Интернета и сохранит, а затем будет извлекать его из кеша.

Виджет также поддерживает дополнительные параметры:

- **placeholder**: виджет, отображаемый во время загрузки изображения
- **errorWidget**: виджет, отображаемый при ошибке загрузки

- **fit**: определяет, как изображение должно вписываться в доступное пространство

## Выполнение практической работы №6

В рамках модернизации приложения для трекинга привычек, разработанного в предыдущих практических работах, была добавлена поддержка загрузки иконок привычек из сети интернет вместо использования предустановленных иконок.

Для обеспечения корректной работы приложения был добавлен пакет `cached_network_image`. Данный пакет позволяет загружать изображения из сети с автоматическим кешированием, что обеспечивает их доступность даже при отсутствии интернет-соединения после первой загрузки.

Добавление зависимости в файл `pubspec.yaml` проиллюстрировано в рисунке 1.

*Листинг 1 — Добавление зависимости `cached_network_image`*

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  # The following adds the Cupertino Icons font to your application.  
  # Use with the CupertinoIcons class for iOS style icons.  
  cupertino_icons: ^1.0.8  
  cached_network_image: ^3.4.1
```

Для реализации новой функциональности были внесены изменения в несколько ключевых файлов проекта.

### Изменения в модели данных

В первую очередь была изменена структура сущности `Habbit`. Ранее для хранения иконки использовался enum `HabbitIcon` с фиксированным набором значений. Этот подход был заменён на использование строкового поля `iconUrl`, которое хранит URL изображения иконки.

Изменения в файле `habbit.dart` показаны в рисунке 2. Удалён enum `HabbitIcon`, а поле `icon` типа `HabbitIcon` заменено на поле `iconUrl` типа `String`. Соответственно были обновлены конструкторы и методы копирования объекта.

## Листинг 2 — Изменённая структура сущности Habbit

```
class HabbitAction {  
    final DateTime occurredOn;  
  
    const HabbitAction({required this.occurredOn});  
}  
  
class Ack extends HabbitAction {  
    Ack({required super.occurredOn});  
}  
  
class Break extends HabbitAction {  
    Break({required super.occurredOn});  
}  
  
class Habbit {  
    final int id;  
    final String name;  
    final String iconUrl;  
    final DateTime createdAt;  
    final int targetDays;  
    final List<HabbitAction> _events;  
  
    factory Habbit({  
        required int id,  
        required String name,  
        required String iconUrl,  
        required DateTime createdAt,  
        required int targetDays,  
    }) {  
        return Habbit._(  
            id: id,  
            name: name ?? this.name,  
            iconUrl: iconUrl ?? this.iconUrl,  
            createdAt: createdAt ?? this.createdAt,  
            targetDays: targetDays ?? this.targetDays,  
            events: events ?? _events,  
        );  
    }  
}  
}
```

Также был изменён метод копирования с модификацией иконки в рисунке 3.

## Листинг 3 — Метод изменения URL иконки

```
Habbit withIconUrl(String iconUrl) {  
    return _copyWith(iconUrl: iconUrl);  
}  
  
Habbit _copyWith({  
    String? name,  
    String? iconUrl,  
    DateTime? createdAt,  
    int? targetDays,  
    List<HabbitAction>? events,  
}) {  
    return Habbit._(  
        id: id,  
        name: name ?? this.name,  
        iconUrl: iconUrl ?? this.iconUrl,  
        createdAt: createdAt ?? this.createdAt,  
        targetDays: targetDays ?? this.targetDays,  
        events: events ?? _events,  
    );  
}
```

## Изменения в форме создания привычки

В форме создания/редактирования привычки был полностью переработан UI выбора иконки. Вместо визуального выбора из предустановленного набора иконок теперь используется текстовое поле для ввода URL изображения.

В рисунке 4 показано изменение контроллеров формы — вместо переменной `selectedIcon` добавлен `iconUrlController` для работы с текстовым полем.

*Листинг 4 — Контроллеры формы с полем URL иконки*

```
class _HabbitFormScreenState extends State<HabbitFormScreen> {
    final _nameController = TextEditingController();
    final _targetDaysController = TextEditingController();
    final _iconUrlController = TextEditingController();

    @override
    void dispose() {
        _nameController.dispose();
        _targetDaysController.dispose();
        _iconUrlController.dispose();
        super.dispose();
    }
}
```

Изменение валидации формы представлено в рисунке 5. Теперь форма считается валидной, если URL иконки не пустой.

*Листинг 5 — Валидация формы*

```
bool get _isValidForm {
    return _nameController.text.isNotEmpty &&
        _iconUrlController.text.isNotEmpty &&
        _targetDaysController.text.isNotEmpty &&
        int.tryParse(_targetDaysController.text) != null &&
        int.parse(_targetDaysController.text) > 0;
```

UI формы теперь содержит простое текстовое поле для ввода URL вместо набора иконок, как показано в рисунке 6.

*Листинг 6 — UI текстового поля для URL иконки*

```
// URL иконки
TextField(
    controller: _iconUrlController,
    decoration: const InputDecoration(
        labelText: 'URL иконки',
        hintText: 'Например: https://example.com/icon.png',
        border: OutlineInputBorder(),
    ),
),
```

**Изменения в отображении привычки**

Самые существенные изменения были внесены в виджет HabbitItem, отвечающий за отображение отдельной привычки в списке. Вместо использования стандартного виджета Icon теперь применяется CachedNetworkImage для загрузки и кеширования изображений.

В рисунке 7 показан импорт пакета cached\_network\_image.

*Листинг 7 — Импорт пакета cached\_network\_image*

```
import 'package:flutter/material.dart';
import 'package:cached_network_image/cached_network_image.dart';
import 'package:practice2/features/entities/habbit.dart';
```

Удалён метод \_getIconData, который ранее преобразовывал enum в IconData. Вместо него теперь используется прямое обращение к полю habbit.iconUrl.

Ключевое изменение — использование виджета CachedNetworkImage, как показано в рисунке 8. Виджет настроен с параметрами:

- **imageUrl**: URL изображения из модели привычки
- **fit**: BoxFit.cover для правильного масштабирования
- **placeholder**: индикатор загрузки, отображаемый во время скачивания
- **errorWidget**: иконка ошибки, показываемая при неудачной загрузке

*Листинг 8 — Использование CachedNetworkImage для отображения иконки*

```
children: [
    // Иконка привычки
    Container(
        width: 56,
        height: 56,
        padding: const EdgeInsets.all(8),
        decoration: BoxDecoration(
            color:
        Theme.of(context).colorScheme.primaryContainer,
            borderRadius: BorderRadius.circular(8),
        ),
        child: ClipRRect(
            borderRadius: BorderRadius.circular(4),
            child: CachedNetworkImage(
                imageUrl: habbit.iconUrl,
                fit: BoxFit.cover,
                placeholder: (context, url) =>
                    const Center(child:
CircularProgressIndicator()),
                errorWidget: (context, url, error) => Icon(
                    Icons.error,
                    color: Theme.of(context).colorScheme.error,
                ),
            ),
        ),
    ),
]
```

*Продолжение листинга 8*

```
        ),  
        ),  
    ),
```

## **Изменения в интерфейсе контроллера**

Интерфейс HabbitsController был обновлён для работы с новой структурой данных. Везде, где ранее использовался тип HabbitIcon, теперь используется String iconName.

Изменения в методах addHabbit и editHabbit показаны в рисунке 9.

*Листинг 9 — Обновлённый интерфейс HabbitsController*

```
void breakHabbit({required int habbitId});  
void removeHabbit({required int habbitId});  
void addHabbit({  
    required String name,  
    required String iconName,  
    required int targetDays,  
});  
List<Habbit> get habbits;  
void showHabbitFormScreen({Habbit? habbit});  
void showHabbitsListScreen();  
void showHabitStatsScreen({required int habbitId});  
void editHabbit({  
    required int habbitId,  
    required String name,  
    required String iconName,  
    required int targetDays,  
});
```

Аналогичные изменения были внесены в реализацию контроллера в классе HabbitsContainer, как показано в рисунке 10 и рисунке 11.

*Листинг 10 — Реализация addHabbit в HabbitsContainer*

```
@override  
void addHabbit({  
    required String name,  
    required String iconName,  
    required int targetDays,  
}) {  
    setState(() {  
        _habbits.add(  
            Habbit(  
                id: widget.nextHabbitId(),  
                name: name,  
                createdAt: widget.currentTimeMillis(),  
                iconName: iconName,  
                targetDays: targetDays,  
            ),
```

## *Продолжение листинга 10*

```
    );  
});
```

## *Листинг 11 — Реализация editHabbit в HabbitsContainer*

```
void editHabbit(  
    required int habbitId,  
    required String name,  
    required String iconUrl,  
    required int targetDays,  
) {  
    _exchange(  
        habbitId: habbitId,  
        mutation: (h) =>  
  
    h.withName(name).withTargetDays(targetDays).withIconUrl(iconUrl),  
    );
```

## **Демонстрация работы приложения**

После внесения всех изменений приложение было протестировано в двух режимах: с подключением к интернету и без него.

На рисунке 1 показан экран добавления новой привычки с полем для ввода URL иконки.

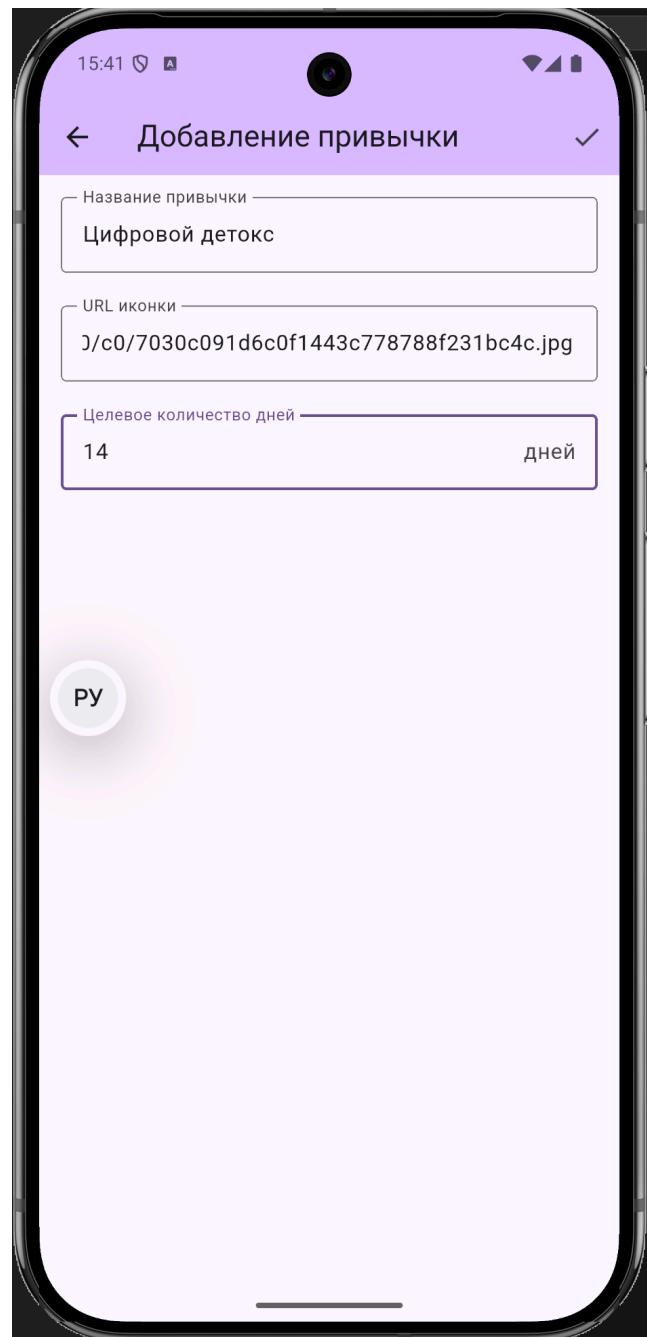


Рисунок 1 — Экран добавления привычки с URL иконки  
рисунке 2 демонстрирует список привычек после добавления нескольких записей. Все иконки успешно загружены из интернета.

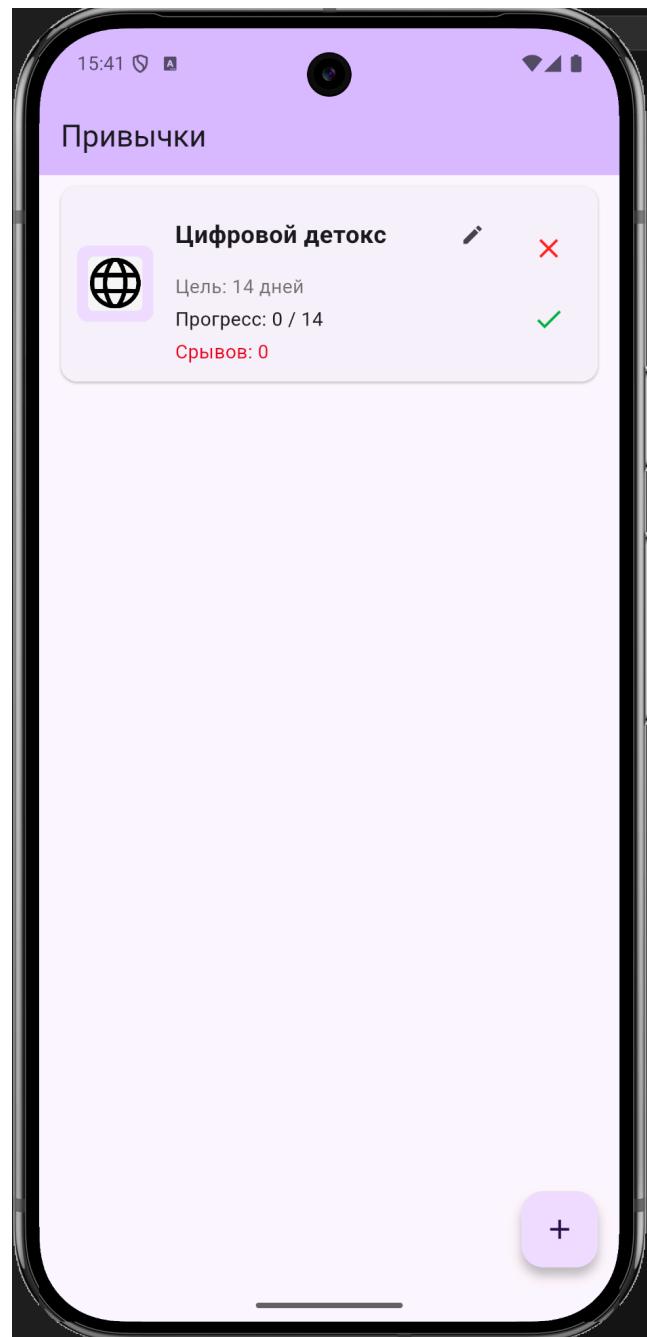


Рисунок 2 — Список привычек с загруженными иконками

Для проверки функциональности кеширования было проведено тестирование с активным подключением к интернету (рисунке 3) и без него (рисунке 4).

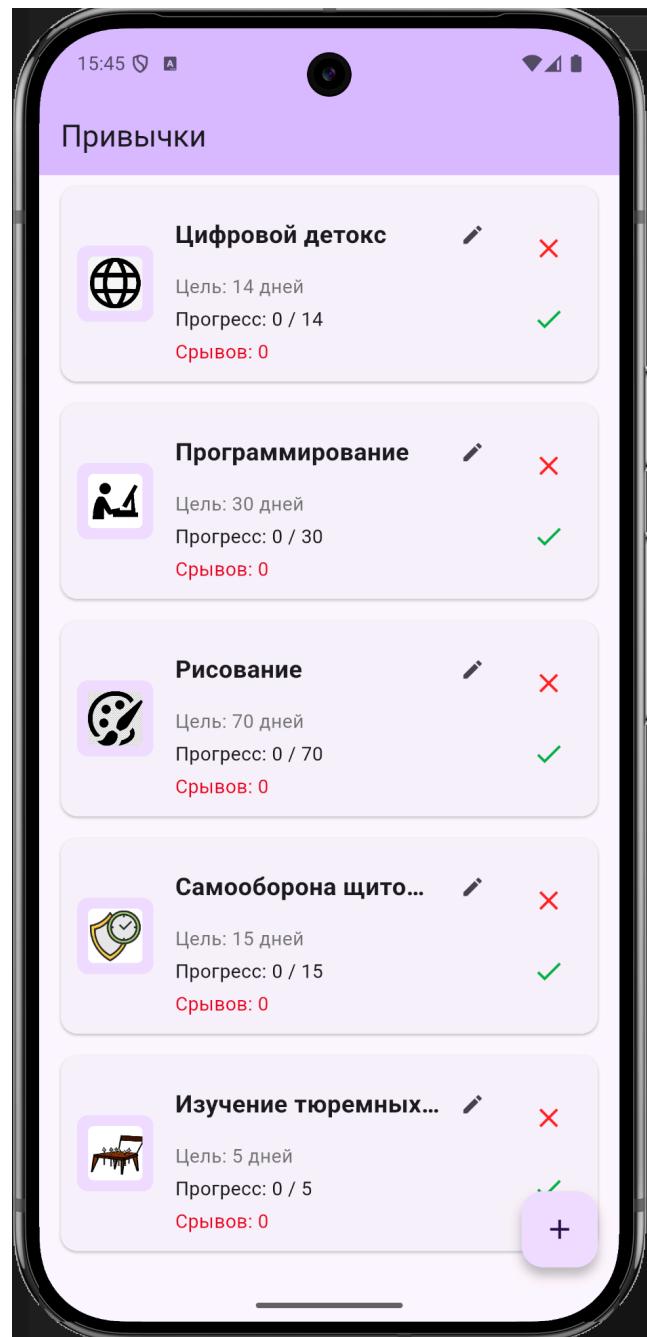


Рисунок 3 — Отображение привычек при наличии интернета

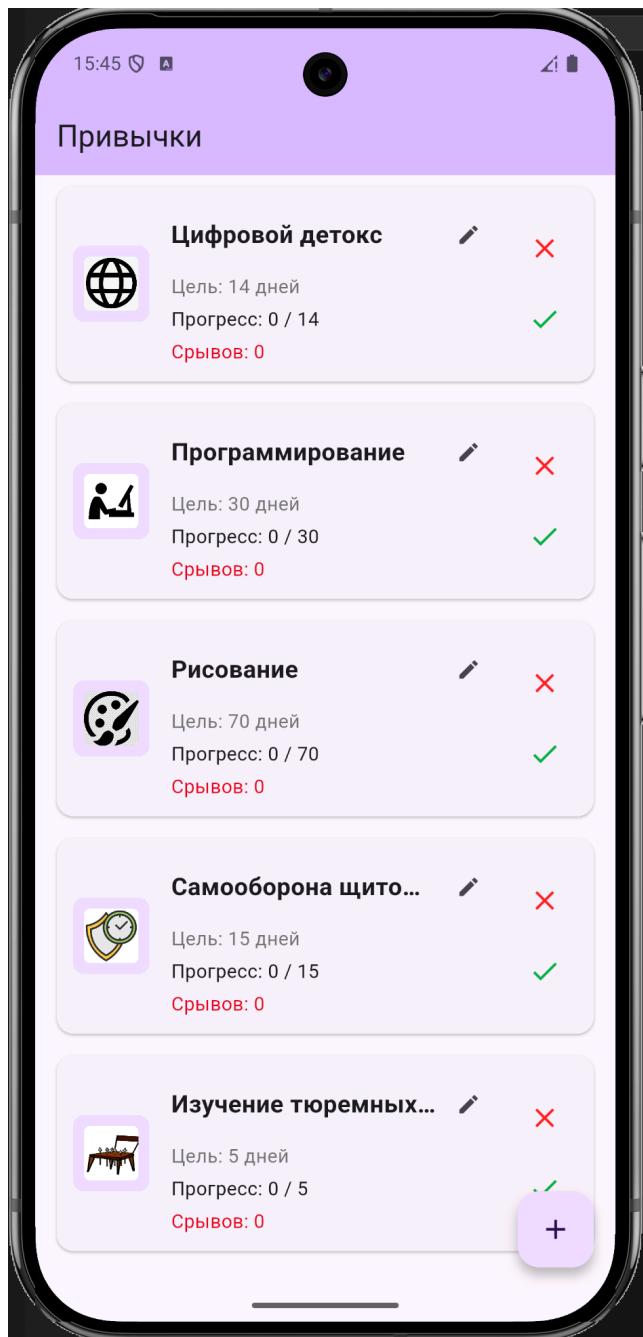


Рисунок 4 — Отображение привычек без подключения к интернету

Как видно из рисунка 4, после первоначальной загрузки изображения остаются доступными даже при отсутствии интернет-соединения. Это подтверждает корректную работу механизма кеширования, реализованного пакетом `cached_network_image`.

Таким образом, приложение стало более гибким — пользователи могут использовать любые изображения для визуализации своих привычек, а не ограничиваться предустановленным набором иконок. При этом сохранена

возможность работы в офлайн-режиме благодаря кэшированию загруженных изображений.

## **Заключение**

В ходе выполнения практической работы было модернизировано приложение для трекинга привычек путём добавления поддержки загрузки пользовательских иконок из сети интернет. Реализация была выполнена с использованием пакета `cached_network_image` версии 3.4.1, который обеспечивает автоматическое кэширование загруженных изображений.

Были внесены изменения в структуру данных (замена enum `HabbitIcon` на `String iconName`), интерфейс пользователя (замена визуального выбора иконок на текстовое поле для ввода URL) и компоненты отображения (использование виджета `CachedNetworkImage` вместо `Icon`).

Тестирование подтвердило корректную работу приложения как при наличии интернет-соединения (загрузка и кэширование новых изображений), так и при его отсутствии (отображение ранее закешированных изображений). Это обеспечивает хороший пользовательский опыт и эффективное использование сетевых ресурсов.