ДИСЦИПЛИНА	Программирование корпоративных систем
ИНСТИТУТ	Институт перспективных технологий и индустриального программирования
КАФЕДРА	Кафедра индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Практические задание
ПРЕПОДАВАТЕЛЬ	Адышкин Сергей Сергеевич
СЕМЕСТР	5 семестр, 2025-2026 гг.

Практическое занятие № 10

Работа с базами данных. Подключение к SQLite (Flutter, sqflite)

Цели занятия

- Подключить Flutter-приложение к локальной базе данных SQLite (пакет `sqflite`).
- Создать таблицы и реализовать базовый CRUD: добавление, чтение, обновление, удаление.
- Настроить класс-помощник для работы с БД и отделить слой данных от UI.
- Отработать миграции схемы и диагностику частых ошибок.

Теоретическая часть

Что такое SQLite и в чём её место в мобильной архитектуре

SQLite — встраиваемая реляционная СУБД, которая живёт в **одном файле** на устройстве пользователя. Не требует сервера и запускается через нативные библиотеки платформы (Android/iOS). Ключевые свойства:

- **Локальность**: данные хранятся в `app.db` внутри песочницы приложения.
- **ACID**: атомарность, согласованность, изолированность, долговечность.
- **SQL-диалект**: таблицы, индексы, транзакции, JOIN, агрегаты.
- **Нулевой DevOps**: нет внешнего инстанса БД.

Где уместна:

- офлайн-режимы, локальные справочники и кэш (например, заметки, задачи, настройки);
- хранение пользовательских данных, которые не обязаны синхронизироваться в «облако»;
- гибридные схемы «локально + синхронизация по сети» (репликация/мердж на своём API).

С чем сравнить:

- **Firebase/Supabase** — облачный бекенд: сетевые задержки, авторизация, масштабируемость, синхронизация «из коробки».
- **SQLite** — мгновенно и офлайн, но без встроенной синхронизации.

Типы данных и модель хранения

SQLite оперирует **типовыми аффинностями** колонок:

- `INTEGER`, `TEXT`, `REAL`, `BLOB`, `NUMERIC`.

Особенности:

- `INTEGER PRIMARY KEY` = «rowid» (уникальный 64-битный идентификатор строки).

- Строки хранятся компактно; размер файла растёт по мере вставок.
- Даты/время обычно сохраняют как:
  - `INTEGER` (Unix time в миллисекундах/секундах) — быстрее сортировки/поиск,
  - либо `TEXT` (ISO8601) — читаемо, удобно для экспорта.

Практика:

- Для меток времени удобно хранить `created_at`, `updated_at` в **миллисекундах** (целое), чтобы не мучиться с форматами и часовыми поясами.

Схема БД и нормализация

Подход к проектированию:

- Начинайте с **минимальной** и **плоской** схемы, затем нормализуйте по необходимости.
- Нормальные формы полезны, но в мобильных приложениях нередко допускается «денормализация ради скорости UI» (меньше JOIN → быстрее и проще код).
- Пример для заметок:
  - `notes(id, title, body, created_at, updated_at)`.
  - Индексы: по полям сортировки/поиска (например, `created_at DESC`, `title`).

Индексы:

- Ускоряют `WHERE`/`ORDER BY`, но замедляют `INSERT/UPDATE`.
- Создавайте индекс **только** там, где есть реальная нагрузка.

Транзакции, журналирование и параллелизм

Транзакции объединяют набор операций в атомарный блок:

```
BEGIN;
  -- несколько INSERT/UPDATE/DELETE
COMMIT; -- или ROLLBACK;
```

В `sqflite`:

```
await db.transaction((txn) async {
  await txn.insert('notes', {...});
  await txn.update('notes', {...}, where: 'id=?', whereArgs: [id]);
});
```

Журналирование:

- **Rollback journal** (по умолчанию) или **WAL (Write-Ahead Logging)**.

  WAL даёт лучшую конкуррентность чтения/записи; в `sqflite` можно включить PRAGMA при открытии (см. раздел «Практики производительности»).

Параллелизм:

- SQLite «однописательна» (одновременно пишет один), но допускает множественные читатели.
- В Flutter важно не блокировать UI-поток: все операции `sqflite` асинхронны (`Future`).

Миграции и версия схемы

`openDatabase(path, version: X, onCreate: ..., onUpgrade: ...)`:

- **version** — номер текущей схемы.
- При первом запуске вызывается `onCreate`.
- При повышении версии — `onUpgrade(oldV, newV)`.

Стратегии миграций:

- **Непотеряные**: `ALTER TABLE ... ADD COLUMN ... DEFAULT ...`.
- **Сложные**: создать временную таблицу → перенести данные → дропнуть старую → переименовать временную.
- **Правило**: каждое изменение схемы — инкремент версии и явный код миграции.

Типичные ошибки:

- «no such table/column» — изменили схему, но **не** подняли `version`.
- «database is locked» — длинные транзакции; держите их короткими.

Паттерны в приложении: DAO, Репозиторий, слой данных

Рекомендуем отделить слои:

- **Модель (Entity)** — plain-класс с `toMap()/fromMap()`.
- **DAO/DBHelper** — сырой SQL, конкретные запросы, миграции.
- **Репозиторий** — бизнес-уровень (например, «получить все заметки»), который UI вызывает, не зная SQL.

Плюсы:

- Тестируемость: можно подменять репозиторий мок-реализацией.
- Повторное использование: тот же UI может жить поверх другой БД/источника.

Основные операции в `sqflite`

- Открытие:
  - путь берём из `path_provider` (`getApplicationDocumentsDirectory()` на iOS/Android),
  - `openDatabase(dbPath, version: ..., onCreate: ..., onUpgrade: ...)`.
- CRUD:
  - `db.insert(table, map)`,
  - `db.query(table, where: 'id=?', whereArgs: [id])`,
  - `db.update(table, map, where: ..., whereArgs: ...)`,
  - `db.delete(table, where: ..., whereArgs: ...)`.
- Сырые запросы:
  - `db.rawQuery('SELECT ...')`, `db.rawInsert(...)`, `db.rawUpdate(...)`, `db.rawDelete(...)`.

- Батчи:
  - `final batch = db.batch(); ...; await batch.commit(noResult: true);` — быстрее при множествах операций.

Производительность и оптимизации

- **Индексы** по часто используемым фильтрам/сортировкам.
- **Batch** для массовых вставок/обновлений.
- **WAL**:

```
await db.execute('PRAGMA journal_mode = WAL;');
await db.execute('PRAGMA synchronous = NORMAL;'); // осторожнее: баланс безопасность/скорость
```

- **Профилирование**: логируйте длительные запросы, следите за временем `Future`.
- **Память**: не тащите все поля в `SELECT *`, ограничивайте колонки.
- **Пагинация** (`LIMIT/OFFSET`) для длинных списков.

Полнотекстовый поиск (FTS)

SQLite поддерживает **FTS4/FTS5** (модуль полнотекстового поиска). Подход:

- Создать виртуальную FTS-таблицу (`CREATE VIRTUAL TABLE notes_fts USING fts5(title, body);`)
- Поддерживать синхронизацию с основной таблицей (триггерами или вручную при вставке/обновлении/удалении).
- Запросы: `SELECT * FROM notes_fts WHERE notes_fts MATCH 'слово*';`

Плюсы:

- Быстрый поиск по текстам.

Минусы:

- Дополнительное место и логика синхронизации.

Надёжность и целостность

- **Транзакции** вокруг связанных изменений.
- **Ограничения**: `NOT NULL`, `UNIQUE`, `CHECK`, `FOREIGN KEY` (включите `PRAGMA foreign_keys = ON;`).
- **Резервное копирование**: копирование файла БД (закрывайте соединение или используйте `VACUUM INTO 'backup.db';` через `rawQuery`).
- **Повреждение файла**: крайне редкое, но возможно при внезапном завершении — WAL снижает риск.

Безопасность и приватность

- Песочница ОС защищает от доступа других приложений.
- Для повышенных требований:
  - **Шифрование** (SQLCipher-сборки; в экосистеме Flutter — отдельные плагины; `sqflite` из коробки не шифрует).
  - Храните только необходимые данные, избегайте секретов в открытом виде.
  - При экспорте/бэкапе фильтруйте персональные поля.

Локаль, время и кодировки

- SQLite хранит `TEXT` в UTF-8/UTF-16, проблем с кириллицей нет.
- Для меток времени придерживайтесь **UTC в миллисекундах** (целое). Преобразование в локальное время — на уровне UI.

Тестирование

- Юнит-тесты слоя данных: создавайте **временную БД в памяти** (`:memory:`) или во временной папке теста.
- Сценарии:
  - `onCreate` создаёт таблицы,

- CRUD сохраняет/читает корректно,
- миграция с `oldV → newV` добавляет нужные колонки и не теряет данные.

Отладка и инспекция

- Инспектор БД на десктопе: DB Browser for SQLite, SQLiteStudio (открывают `.db` файл, удобно смотреть таблицы/индексы).
- На эмуляторе Android: можно вытянуть файл через `adb pull` (в учебных целях) или использовать внутренние страницы разработчика (если встроите).

Ограничения и подводные камни

- **Однописательность**: одна запись за раз; избегайте долгих транзакций на главном сценарии UI.
- **Схемные ловушки**: перед удалением/переименованием колонок готовьте миграции.
- **Размер файла**: при массовых удалениях выполняйте `VACUUM` (вручную/по событию), чтобы уплотнить файл.
- **Связи**: внешние ключи требуют `PRAGMA foreign_keys=ON;` (включайте при открытии).

Шаблоны SQL для справочника

Создание таблицы:

```
CREATE TABLE notes (
  id         INTEGER PRIMARY KEY AUTOINCREMENT,
  title      TEXT NOT NULL,
  body       TEXT NOT NULL,
  created_at INTEGER NOT NULL,
  updated_at INTEGER NOT NULL
);
CREATE INDEX idx_notes_created_at ON notes(created_at DESC);
```

Изменение схемы (миграция v1 → v2):

```
ALTER TABLE notes ADD COLUMN is_favorite INTEGER NOT NULL DEFAULT 0;
CREATE INDEX idx_notes_favorite ON notes(is_favorite);
```

Выборки:

```
-- последние заметки
SELECT id, title, body FROM notes ORDER BY created_at DESC LIMIT 50;

-- поиск по заголовку (без FTS)
SELECT id, title FROM notes WHERE title LIKE '%' || ? || '%' ORDER BY created_at DESC;
```

Транзакция вставок:

```
BEGIN;
INSERT INTO notes(title, body, created_at, updated_at) VALUES(?, ?, ?, ?);
-- ещё операции...
COMMIT;
```

Практики «боевого» применения в Flutter (sqflite)

- Открывайте БД **один раз** и переиспользуйте соединение (Singleton в `DBHelper`).
- Любые длительные операции — **await** + прогресс-индикаторы в UI.
- Для больших списков: `ListView.builder`, пагинация на уровне SQL (LIMIT/OFFSET).
- При изменении схемы:
  - увеличили `version`,
  - написали `onUpgrade` с пошаговыми `if (oldV < 2) {...} if (oldV < 3) {...}`,
  - протестировали апгрейд на копии реальной БД.
- Для полнотекстового поиска: выделяйте FTS-таблицу и триггеры синхронизации.

- Для многомодульных приложений: выносите слой данных в отдельный пакет/подпапку с DAO и репозиторием; не смешивайте SQL в виджетах.

Сравнение с альтернативами в Flutter

- **Drift (moor)**: безопасный типизированный слой поверх SQLite (генерация кода, миграции декларативно). Проще тесты, сложнее старт.
- **Hive/Isar**: NoSQL/объектные хранилища, очень быстрые KV/объектные операции, но нет SQL/JOIN; удобны для кэшей/настроек/иерархий объектов.
- **ObjectBox**: высокопроизводительное объектное хранилище с relation-концепцией и кросс-платформенной поддержкой.

Выбор делаем по требованиям: нужен SQL/JOIN и переносимость — SQLite/sqflite; нужна скорость/объектная модель — Isar/Hive; нужна типобезопасность SQL — Drift.

Практическая часть

Итог работы

Мини-приложение **Notes SQLite**: список заметок (заголовок + текст) с CRUD, хранение в локальной БД.

0) Подготовка проекта

```
flutter create notes_sqlite_app
cd notes_sqlite_app
```

В `pubspec.yaml` добавьте зависимости:

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.3.3
  path: ^1.9.0
  path_provider: ^2.1.4
```

1) Структура каталогов (минимум)

```
lib/
  main.dart
  data/
    db_helper.dart      // открытие БД, миграции, CRUD
  models/
    note.dart           // модель заметки
  pages/
        notes_page.dart          //  UI:  список,  диалоги добавления/редактирования
```

Этот подход согласуется с ранее отработанной логикой экранов и списков.

2) Модель данных

`lib/models/note.dart`

```
class Note {
  final int? id;       // null до вставки
  final String title;
  final String body;
  final DateTime createdAt;
  final DateTime updatedAt;
```

```
      Note({
        this.id,
        required this.title,
        required this.body,
        required this.createdAt,
        required this.updatedAt,
      });

       Note copyWith({int? id, String? title, String? body,
DateTime? createdAt, DateTime? updatedAt}) {
        return Note(
          id: id ?? this.id,
          title: title ?? this.title,
          body: body ?? this.body,
          createdAt: createdAt ?? this.createdAt,
          updatedAt: updatedAt ?? this.updatedAt,
        );
      }

      factory Note.fromMap(Map<String, Object?> map) => Note(
            id: map['id'] as int?,
            title: map['title'] as String? ?? '',
            body: map['body'] as String? ?? '',
                                                  createdAt:
DateTime.fromMillisecondsSinceEpoch(map['created_at'] as int),
                                                  updatedAt:
DateTime.fromMillisecondsSinceEpoch(map['updated_at'] as int),
          );

      Map<String, Object?> toMap() => {
            'id': id,
            'title': title,
            'body': body,
            'created_at': createdAt.millisecondsSinceEpoch,
            'updated_at': updatedAt.millisecondsSinceEpoch,
```

```
      };
  }
```

3) Класс работы с БД (открытие, миграции, CRUD)

lib/data/db_helper.dart

```
import 'dart:async';
import 'package:path/path.dart' as p;
import 'package:path_provider/path_provider.dart';
import 'package:sqflite/sqflite.dart';
import '../models/note.dart';

class DBHelper {
  static const _dbName = 'app.db';
  static const _dbVersion = 1; // увеличивайте при изменении
схемы
  static Database? _db;

  static const notesTable = 'notes';

  static Future<Database> _open() async {
    if (_db != null) return _db!;
    final docs = await getApplicationDocumentsDirectory();
    final dbPath = p.join(docs.path, _dbName);
    _db = await openDatabase(
      dbPath,
      version: _dbVersion,
      onCreate: (db, version) async {
        await db.execute('''
          CREATE TABLE $notesTable(
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,
            body TEXT NOT NULL,
            created_at INTEGER NOT NULL,
            updated_at INTEGER NOT NULL
          );
        ''');
```

```
            await db.execute('CREATE INDEX idx_notes_created_at
ON $notesTable(created_at DESC);');
          },
          onUpgrade: (db, oldV, newV) async {
            // Пример миграции:
              // if (oldV < 2) { await db.execute('ALTER TABLE
$notesTable ADD COLUMN ...'); }
          },
        );
        return _db!;
      }

      // CREATE
      static Future<int> insertNote(Note note) async {
        final db = await _open();
                  return   db.insert(notesTable,   note.toMap(),
conflictAlgorithm: ConflictAlgorithm.abort);
      }

      // READ all (сортировка по created_at DESC)
      static Future<List<Note>> fetchNotes() async {
        final db = await _open();
            final  rows  =  await  db.query(notesTable,  orderBy:
'created_at DESC');
        return rows.map((m) => Note.fromMap(m)).toList();
      }

      // UPDATE
      static Future<int> updateNote(Note note) async {
        final db = await _open();
        return db.update(
          notesTable,
          note.toMap(),
          where: 'id = ?',
          whereArgs: [note.id],
          conflictAlgorithm: ConflictAlgorithm.abort,
```

```
        );
      }

      // DELETE
      static Future<int> deleteNote(int id) async {
        final db = await _open();
        return db.delete(notesTable, where: 'id = ?', whereArgs:
[id]);
      }
    }
```

4) UI: список + диалоги добавления/редактирования

lib/pages/notes_page.dart

```
    import 'package:flutter/material.dart';
    import '../data/db_helper.dart';
    import '../models/note.dart';

    class NotesPage extends StatefulWidget {
      const NotesPage({super.key});
      @override
      State<NotesPage> createState() => _NotesPageState();
    }

    class _NotesPageState extends State<NotesPage> {
      late Future<List<Note>> _future;

      @override
      void initState() {
        super.initState();
        _reload();
      }

          void _reload() => setState(() => _future =
DBHelper.fetchNotes());

      Future<void> _createDialog() async {
```

```
        final titleCtrl = TextEditingController();
        final bodyCtrl  = TextEditingController();
        final ok = await showDialog<bool>(
          context: context,
             builder: (_) => _EditDialog(titleCtrl: titleCtrl,
bodyCtrl: bodyCtrl, title: 'Новая заметка'),
        );
        if (ok == true) {
          final now = DateTime.now();
                         await   DBHelper.insertNote(Note(title:
titleCtrl.text.trim(),  body:  bodyCtrl.text.trim(),  createdAt:
now, updatedAt: now));
          _reload();
        }
      }

      Future<void> _editDialog(Note n) async {
        final titleCtrl = TextEditingController(text: n.title);
        final bodyCtrl  = TextEditingController(text: n.body);
        final ok = await showDialog<bool>(
          context: context,
             builder: (_) => _EditDialog(titleCtrl: titleCtrl,
bodyCtrl: bodyCtrl, title: 'Редактировать'),
        );
        if (ok == true) {
          final updated = n.copyWith(
            title: titleCtrl.text.trim(),
            body: bodyCtrl.text.trim(),
            updatedAt: DateTime.now(),
          );
          await DBHelper.updateNote(updated);
          _reload();
        }
      }

      Future<void> _delete(Note n) async {
```

```
        await DBHelper.deleteNote(n.id!);
        if (!mounted) return;
                ScaffoldMessenger.of(context).showSnackBar(const
SnackBar(content: Text('Удалено')));
        _reload();
      }

      @override
      Widget build(BuildContext context) {
        return Scaffold(
          appBar: AppBar(title: const Text('Notes SQLite')),
           floatingActionButton: FloatingActionButton(onPressed:
_createDialog, child: const Icon(Icons.add)),
          body: FutureBuilder<List<Note>>(
            future: _future,
            builder: (context, snap) {
                 if (snap.hasError) return const Center(child:
Text('Ошибка загрузки'));
                if (!snap.hasData)   return const Center(child:
CircularProgressIndicator());
              final notes = snap.data!;
                 if (notes.isEmpty) return const Center(child:
Text('Пока нет заметок. Нажмите +'));
              return ListView.separated(
                padding: const EdgeInsets.all(12),
                itemCount: notes.length,
                          separatorBuilder:  (_,  __)  =>  const
SizedBox(height: 8),
                itemBuilder: (_, i) {
                  final n = notes[i];
                  return Dismissible(
                    key: ValueKey(n.id),
                                background:  Container(color:
Theme.of(context).colorScheme.error.withOpacity(.1)),
                    onDismissed: (_) => _delete(n),
                    child: Card(
```

```
                    child: ListTile(
                          title: Text(n.title.isEmpty ? '(без
названия)'     :     n.title,     maxLines:     1,     overflow:
TextOverflow.ellipsis),
                          subtitle: Text(n.body, maxLines: 2,
overflow: TextOverflow.ellipsis),
                      onTap: () => _editDialog(n),
                             trailing: IconButton(icon: const
Icon(Icons.delete_outline), onPressed: () => _delete(n)),
                    ),
                  ),
                );
              },
            );
          },
        ),
      );
    }
  }

  class _EditDialog extends StatelessWidget {
    final TextEditingController titleCtrl;
    final TextEditingController bodyCtrl;
    final String title;
      const  _EditDialog({required  this.titleCtrl,  required
this.bodyCtrl, required this.title});

    @override
    Widget build(BuildContext context) {
      return AlertDialog(
        title: Text(title),
             content:  Column(mainAxisSize:  MainAxisSize.min,
children: [
            TextField(controller: titleCtrl, decoration: const
InputDecoration(labelText: 'Заголовок')),
```

```
              TextField(controller: bodyCtrl,  decoration: const
InputDecoration(labelText: 'Текст')),
            ]),
            actions: [
               TextButton(onPressed: () => Navigator.pop(context,
false), child: const Text('Отмена')),
              FilledButton(onPressed: () => Navigator.pop(context,
true), child: const Text('Сохранить')),
            ],
          );
        }
      }
      lib/main.dart
      import 'package:flutter/material.dart';
      import 'pages/notes_page.dart';

      void main() => runApp(const NotesApp());

      class NotesApp extends StatelessWidget {
        const NotesApp({super.key});
        @override
        Widget build(BuildContext context) {
          return MaterialApp(
            title: 'Notes SQLite',
            theme: ThemeData(useMaterial3: true),
            home: const NotesPage(),
          );
        }
      }
```

5) Контрольные точки

1. Проект собирается и запускается на устройстве/эмуляторе.
2. При первом старте создаётся таблица `notes` без ошибок.
3. Кнопка «+» открывает диалог и сохраняет запись в БД.

4. Список показывает данные из БД; редактирование обновляет запись.

5. Удаление через иконку/свайп удаляет строку из БД. *(UI-паттерны списков и взаимодействий соответствуют ранее отработанным практикам)*.

6) Что сдаём (контрольные задания)

- Скриншот приложения с пустым списком (первый запуск).
- Скриншот после добавления заметки.
- Скриншот окна редактирования и итоговой записи.
- Скриншот после удаления (запись исчезла).
- Текстом (0,5–1 стр.): где хранится файл БД, как устроены таблица и индексы, как реализованы CRUD.

7) Требования к отчёту

- Объём: 2–4 страницы + скриншоты контрольных этапов.
- Структура: цель → ход работы (шаги, ключевые фрагменты кода) → результаты → выводы/проблемы и решения.
- По желанию: залить код в Git и приложить ссылку/скриншоты.

8) Частые проблемы и быстрые решения

- **«no such table: notes»** — вы изменили SQL-схему, но не подняли `version`. Решение: увеличить `_dbVersion`, описать миграцию в `onUpgrade`, либо удалить приложение (сбросить БД) на эмуляторе.
- **Блокировки БД** — не держите открытыми «долгие» транзакции; используйте `await` корректно.
- **Кодировка/эмодзи** — храните текст как `TEXT`, проблем обычно нет.
- **Производительность списков** — при больших объёмах используйте пагинацию/поиск и ListView.builder, как делали в ПЗ 5.

9) Дополнительно (+1 балл по желанию)

- Поиск по заголовку (поле ввода в AppBar, `LIKE` с индексом по `title`).

- Миграция схемы: добавить колонку `is_favorite INTEGER NOT NULL DEFAULT 0`, поднять `version` и реализовать `onUpgrade`.
- Вынести слой данных в **репозиторий** и покрыть его модульными тестами.

Контрольные вопросы

1. В чём ключевые отличия локальной SQLite от облачных БД (Firebase/Supabase) по архитектуре и сценариям?
2. Как устроено открытие БД и зачем нужна версия (`version`) и `onUpgrade`?
3. Почему важно отделять слой данных (DAO/репозиторий) от UI?
4. Какие типы данных SQLite вы использовали в таблице `notes` и почему?
5. Как вы реализовали чтение списка и обновление UI после изменения данных? (подсказка: Future + setState / перезагрузка списка)