

Методические указания к Практической работе №8 по дисциплине «Разработка кроссплатформенных мобильных приложений»

Тема работы: Реализация слоя данных Flutter-приложения с использованием локальной БД Drift, паттерна Repository и DI (GetIt)

Цель работы: освоить принципы реализации слоя данных в архитектуре Flutter-приложения с использованием локального хранилища (Drift), паттерна Repository, контейнера внедрения зависимостей (GetIt).

План практической работы:

- Архитектура приложения. Слой данных (Data Layer). Паттерн репозитория.
- Внедрение зависимостей (Dependency Injection (DI)) с использованием GetIt
- Локальное хранение данных на устройстве
- Выполнение практической работы №8 в соответствии с заданием

Последовательность выполнения практической работы:

1. Архитектура приложения. Слой данных (Data Layer)

Обычно архитектура мобильного приложения строится в **виде** многоуровневой (многослойной) модели, каждый слой которой отвечает за определённый функционал.

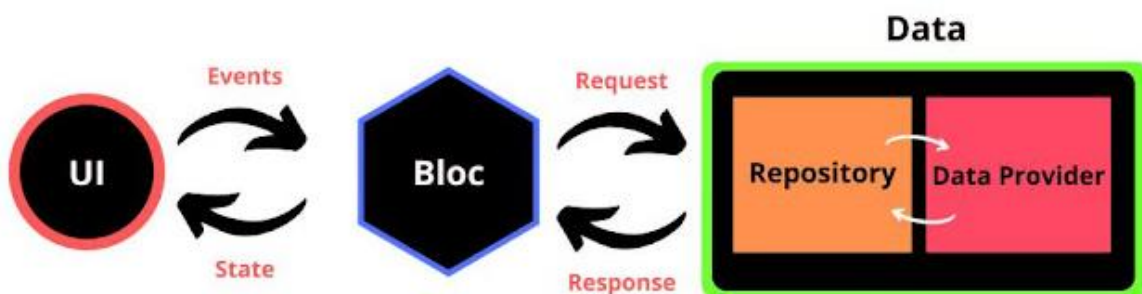
UI Layer (Слой представления) отвечает за отображение данных и взаимодействие с пользователем.

Слой логики (Logic layer, Domain Layer) реализует основную бизнес-логику приложения и обеспечивает взаимодействие между слоем данных и UI-слоем. Этот слой не является обязательным — его стоит добавлять только в том случае, если в приложении есть сложная логика на стороне клиента. Многие приложения выполняют лишь базовые операции — отображают данные, позволяют пользователю изменять их и сохранять (так называемые CRUD-операции (Create, Read, Update, Delete)). В таких случаях этот слой можно опустить.

Слой данных (Data Layer) — отвечает за получение, хранение и изменение данных приложения (БД, API, файлы и т. д.). UI и BLoC не должны знать, как именно хранятся данные — только: какие методы доступны.

Слой данных содержит реализацию источников данных и репозитория, соответствующих интерфейсам из слоя domain. Здесь происходит взаимодействие с внешними API, парсинг данных и преобразование DTO-моделей в доменные модели.

Слой данных состоит из двух частей: репозиторий и поставщик данных.



Паттерн Repository — прослойка между бизнес-логикой и конкретным хранилищем.

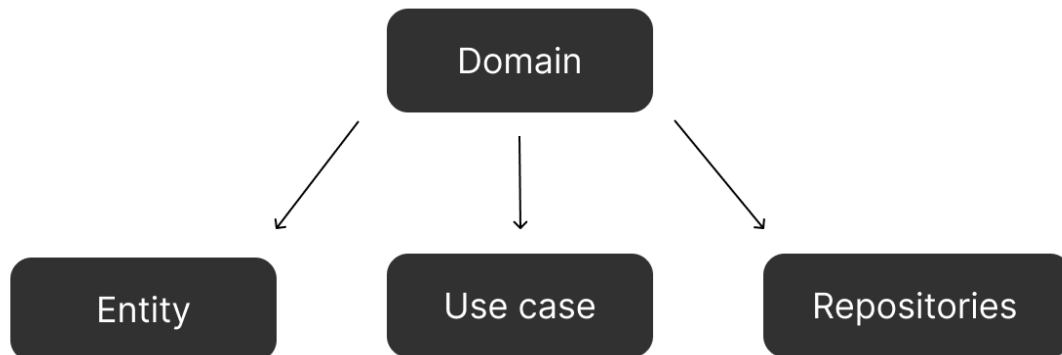
Шаблон репозитория является наиболее подходящим для:

- взаимодействия с REST API;
- взаимодействия с локальными или удаленными базами данных (например, Hive, Firestore и т. д.)
- взаимодействия с API, специфичными для устройств (например, разрешения, камера, местоположение и т. д.).

Таким образом, уровень репозитория выступает в качестве посредника между источниками данных (такими как API, базы данных или локальное хранилище) и уровнем бизнес-логики. Он абстрагируется от деталей того, как извлекаются данные, и предоставляет чистый API для бизнес-логики для извлечения данных (например: преобразование необработанного файла в некую модель).

Поставщик данных извлекает необработанные данные из разных источников данных (например, различных API, баз данных, сети, общих настроек и т. д.). Внутри этого класса могут быть методы GET, POST, DELETE и т.д.

Слой логики (Logic layer, Domain Layer) содержит основные бизнес-правила и логику приложения. Этот слой полностью независим от других слоев, что позволяет легко тестировать и обслуживать его. Доменный уровень включает в себя сущности (Entity), репозитории (Repositories) и варианты использования (Use_case).



Сущности (Entity) являются основными объектами приложения, инкапсулирующими наиболее общие и высокоуровневые бизнес-правила. Они представляют собой простые классы Dart, определяющие свойства и поведение основных бизнес-объектов.

Пример кода для User Entity:

```
class User {
  final String id;
  final String name;
  final String email;

  User({required this.id, required this.name, required this.email});
}
```

Репозитории определяют интерфейсы для операций с данными. Они абстрагируют уровень данных, гарантируя, что уровень предметной области не зависит от какого-либо конкретного источника данных или деталей реализации.

Пример кода для интерфейса пользовательского репозитория:

```
abstract class UserRepository {
  Future<User> getUserDetails(String id);
}
```

Use_case (сценарии использования (или интеракторы)) содержат специфичную для приложения бизнес-логику. Они определяют операции, которые могут быть выполнены с сущностями, и взаимодействуют с репозиториями для выполнения этих операций. Каждый вариант использования обычно соответствует определенному действию пользователя или требованию к функции.

Пример кода для класса GetUserDetails:

```
class GetUserDetails {
  final UserRepository repository;
  GetUserDetails(this.repository);

  Future<User> findUser(String id) {
    return repository.getUserDetails(id);
  }
}
```

Уровень данных отвечает за обработку операций с данными, включая получение данных из удаленных или локальных источников и преобразование необработанных данных в сущности предметной области. Этот слой включает модели, источники данных и реализации репозитория.

Модели – это представления данных, используемые для передачи данных между уровнем данных и другими уровнями приложения. Они часто сопоставляются со структурами JSON, полученными от API или баз данных.

Пример кода для UserModel:

```
class UserModel {
  final String id;
  final String name;
  final String email;

  UserModel({required this.id, required this.name, required this.email});

  factory UserModel.fromJson(Map<String, dynamic> json) {
    return UserModel(
      id: json['id'],
      name: json['name'],
      email: json['email'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'name': name,
      'email': email,
    };
  }
}
```

Источники данных отвечают за получение данных из удаленных или локальных источников. Они определяют методы извлечения и хранения данных и абстрагируются интерфейсами, что упрощает тестирование и замену.

Пример кода для интерфейса UserRemoteDataSource:

```
abstract class UserRemoteDataSource {
  Future<UserModel> fetchUserDetails(String id);
}
```

Пример кода для реализации UserRemoteDataSource:

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

class UserRemoteDataSourceImpl implements UserRemoteDataSource {
  final http.Client client;
  UserRemoteDataSourceImpl({required this.client});

  @override
  Future<UserModel> fetchUserDetails(String id) async {
    final response = await client.get(Uri.parse('https://api.example.com/users/$id'));

    if (response.statusCode == 200) {
      return UserModel.fromJson(json.decode(response.body));
    } else {
      throw Exception('Failed to load user details');
    }
  }
}
```

Реализации репозитория

Реализации репозитория – это конкретные классы, реализующие интерфейсы репозитория, определенные на уровне предметной области. Они используют источники данных для получения и сохранения данных и преобразования моделей данных в сущности предметной области.

Пример кода для реализации UserRepository:

```
class UserRepositoryImpl implements UserRepository {
    final UserRemoteDataSource remoteDataSource;
    UserRepositoryImpl({required this.remoteDataSource});

    @override
    Future<User> getUserDetails(String id) async {
        final userModel = await remoteDataSource.fetchUserDetails(id);
        return User(id: userModel.id, name: userModel.name, email: userModel.email);
    }
}
```

Такая структура слоев обеспечивает четкое разделение обязанностей, гарантируя, что операции с данными являются модульными, тестируемыми и независимыми от основной бизнес-логики.

2. Внедрение зависимостей (Dependency Injection (DI)) с использованием GetIt

DI (Dependency Injection) – это набор паттернов в разработке ПО, позволяющий писать слабо связанный код, который будет легче расширять и поддерживать. Согласно ему объект, нуждающийся в некоторых зависимостях (то есть других объектах, которые он использует в своей работе), должен получать их извне вместо того, чтобы создавать самостоятельно.

Внедрение зависимостей — это механизм, при котором один объект получает свои зависимости извне (через конструктор, фабрику или DI-контейнер), а не создаёт их самостоятельно.

Внедрение зависимостей:

- Вместо того, чтобы создавать объекты внутри класса или метода, эти объекты "внедряются" извне.
- Классу не нужно знать, как создавать объекты, от которых он зависит, ему просто нужно знать, как их использовать.
- Это создает код, который легче тестировать и который более удобен в обслуживании.

DI-контейнеры (GetIt, Provider, Riverpod)

Когда приложение маленькое (1–2 экрана), можно спокойно использовать внедрение зависимостей через конструктор:

```
final repository = TermsRepository();
final bloc = TermsBloc(repository);
```

Но когда в приложении более 10 экранов, несколько BLoC'ов, репозиториев, сервисов авторизации, кеша, настроек и т.д, тогда:

- в каждом месте: final api = ApiClient(); final repository = UserRepo(api); final bloc = UserBloc(repository);
- сложно поменять реализацию (например, с моковой на реальную)
- тяжело тестировать (класс сам создаёт зависимости - не подменишь)

Пакет GetIt

GetIt — сервис-локатор: глобальный «контейнер», в который мы «регистрируем» зависимости, а потом достаём в любом месте.

get_it — это простой, типобезопасный сервисный локатор, который предоставляет O(1) доступ к объектам из любой точки приложения — без необходимости BuildContext, без генерации кода.

GetIt — это service locator, то есть глобальный контейнер, где хранятся зависимости.

Использование пакета get_it

В файл pubspec.yaml проекта добавить пакет get_it в качестве зависимости:

dependencies:

get_it: ^9.0.5

Типы регистрации зависимостей:

Singleton — создается один раз и на протяжении всего ЖЦ приложения существует только один экземпляр. Идеально подходит для сервисов, которые поддерживают состояние.

LazySingleton — создается не сразу при регистрации, а при первом вызове. Откладывает инициализацию до тех пор, пока она не понадобится.

Factory — создается каждый раз новый экземпляр. Отлично подходит для служб без отслеживания состояния или объектов с коротким временем жизни.

Алгоритм применения GetIt в Flutter-проектах

1. Создать DI-контейнер (например, файл lib/di.dart)

```
import 'package:get_it/get_it.dart';
```

```
final getIt = GetIt.instance;
```

2. Зарегистрировать зависимости

Например, в функции `setupDI()` или `configureDependencies()`, которую вызывают 1 раз — из `main()`.

Примеры регистраций:

Singleton — создаётся один раз на всё приложение:

```
getIt.registerLazySingleton<AuthService>(() => AuthService());
```

Factory — создаётся каждый раз при запросе:

```
getIt.registerFactory<LoginBloc>(() => LoginBloc(getIt()));
```

AsyncSingleton — для БД, например Hive/SharedPreferences:

```
getIt.registerSingletonAsync<DatabaseService>() async {  
  final db = DatabaseService();  
  await db.init();  
  return db;  
});
```

3. Инициализировать DI перед запуском приложения

В `main.dart`:

```
void main() {  
  setupDI(); // регистрация зависимостей  
  runApp(const MyApp());  
}
```

4. Получать зависимости через `getIt<K>()` внутри виджетов / сервисов:

```
final api = getIt<ApiService>();
```

Внутри Bloc:

```
class TermsBloc extends Bloc<TermsEvent, TermsState> {  
  final TermsRepository repository = getIt();  
}
```

Внутри фабрики BlocProvider:

```
BlocProvider(  
  create: (_) => TermsBloc(getIt<TermsRepository>()),  
)
```

Пакет Injectable — это кодогенератор поверх GetIt, который автоматически регистрирует все зависимости по аннотациям.

Связка GetIt + Injectable (Injection) — одна из самых популярных в Flutter и вообще в Dart-проектах. Она используется для автоматизации Dependency Injection, чтобы не писать вручную длинный код регистрации зависимостей, когда их много в приложении.

Установка в pubspec.yaml:

dependencies:

```
  get_it: ^9.0.5
```

```
  injectable: ^2.6.0
```

dev_dependencies:

```
  build_runner: ^2.10.2
```

```
  injectable_generator: ^2.9.1
```

Использование

1) Требуется глобальный файл для подготовки ресурсов, которые будут использоваться (файл lib/di.dart)

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
final getIt = GetIt.instance;
@InjectableInit()
void configureDependencies() => $initGetIt(getIt);
```

Основные аннотации:

@injectable — обычный класс, создаётся через registerFactory (по умолчанию).

@lazySingleton — singleton, создаётся при первом запросе.

@singleton — singleton, создаётся сразу.

2) Генерация нового файла для get_it. Чтобы сгенерировать файл, необходимо выполнить следующую команду:

```
flutter pub run build_runner build --delete-conflicting-outputs
```

Этот код генерирует новый файл с именем di.config.dart, который будет включать все зависимости для всех вариантов использования.

3) Затем мы можем добавить configureDependencies() к основной функции. Это позволяет запускать службы в первую очередь, если есть какие-либо сгенерированные токены или асинхронные функции, которые необходимо разрешить перед запуском приложения:

```
void main() {
  configureDependencies();
  runApp(MyApp());
}
```

Injectable — это надстройка над GetIt, которая по аннотациям (@injectable, @lazySingleton, ...) автоматически генерирует код регистрации зависимостей в GetIt вместо ручного getIt.register....

Пример приложения с управлением состоянием с помощью паттерна BLoC (пакет flutter_bloc) без слоя данных с внедрением зависимостей (GetIt + Injectable).

Приложение состоит из 3-х экранов:

- 1) Стартовый экран (кнопка «Показать»)
- 2) Экран списка IT-терминов со строкой поиска по термину
- 3) Экран детализации термина

Данные (список терминов) хранятся в файле data/terms_repository.dart в виде коллекции List<Term>, где Term — класс для описания терминов.

Выполнение:

1) В файл pubspec.yaml проекта добавить пакеты get_it и injectable в качестве зависимостей:

dependencies:

get_it: ^9.0.5

injectable: ^2.6.0

dev_dependencies:

injectable_generator: ^2.9.1

build_runner: ^2.10.2

2) Изменение файла конфигурации DI (lib/di.dart): создаем DI-контейнер getIt и добавляем аннотацию @InjectableInit(), которая сообщает библиотеке injectable, что необходимо сгенерировать код (в файле di.config.dart) и подготовить функцию init(), которая автоматически регистрирует все зависимости в GetIt. Функция init() находит все классы, помеченные аннотациями и регистрирует их реализации в GetIt.

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'di.config.dart';

final getIt = GetIt.instance;

@InjectableInit()
void configureDependencies() => getIt.init();
```

3) Помечаем классы аннотациями (файл lib/data/terms_repository.dart и файл lib/bloc/terms_bloc.dart):
Файл lib/data/terms_repository.dart (аннотация @lazySingleton)

```
import 'package:injectable/injectable.dart';
import '../models/term.dart';

@lazySingleton
class TermsRepository {
  Future<List<Term>> getTerms({bool forceRefresh = false}) async {
    // здесь может быть HTTP-запрос, БД
    await Future.delayed(const Duration(milliseconds: 300));
    return [
      Term(
        title: 'API',
        shortDescription: 'Интерфейс для взаимодействия программ',
        fullDescription: 'API (Application Programming Interface).',
      ),
      Term(
        title: 'HTTP',
        shortDescription: 'Протокол передачи гипертекста',
        fullDescription: 'HTTP (HyperText Transfer Protocol).',
      ),
      Term(
        title: 'JSON',
        shortDescription: 'Текстовый формат обмена данными',
        fullDescription: 'JSON (JavaScript Object Notation).',
      ),
    ];
  }

  Future<List<Term>> searchTerms(String termin) async {
    final all = await getTerms();
    return all
      .where(
        (t) => t.title.toLowerCase().contains(termin.toLowerCase()),
      )
      .toList();
  }
}
```

Файл lib/bloc/terms_bloc.dart (аннотация @injectable)

```
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:terms_bloc_example/data/terms_repository.dart';
import 'package:terms_bloc_example/models/term.dart';
import 'package:meta/meta.dart';
import 'package:injectable/injectable.dart';

part 'terms_event.dart';
part 'terms_state.dart';

@injectable
class TermsBloc extends Bloc<TermsEvent, TermsState> {
  final TermsRepository repository;

  TermsBloc(this.repository) : super(TermsInitial()) {
    on<LoadTermsEvent>(_onLoadTerms);
    on<SearchTermsEvent>(_onSearchTerms);
  }
}
```

```

Future<void> _onLoadTerms(
  LoadTermsEvent event,
  Emitter<TermsState> emit,
) async {
  emit(TermsLoading());
  try {
    final terms = await repository.getTerms();
    emit(TermsLoaded(
      allTerms: terms,
      visibleTerms: terms,
    ));
  } catch (e) {
    emit(TermsError('Не удалось загрузить термины'));
  }
}

Future<void> _onSearchTerms(
  SearchTermsEvent event,
  Emitter<TermsState> emit,
) async {
  final current = state;
  if (current is! TermsLoaded) return;

  final find_term = event.find_term.toLowerCase();

  final filtered = current.allTerms.where((term) {
    return term.title.toLowerCase().contains(find_term);
  }).toList();

  emit(current.copyWith(
    find_term: event.find_term,
    visibleTerms: filtered,
  ));
}
}

```

4) Генерируем DI-код

В терминале:

flutter pub run build_runner build --delete-conflicting-outputs

Появится файл:

lib/di.config.dart

Здесь теперь автоматически зарегистрированы все классы.

```

// GENERATED CODE - DO NOT MODIFY BY HAND
// dart format width=80

// *****
// InjectableConfigGenerator
// *****

// ignore_for_file: type=lint
// coverage:ignore-file

// ignore_for_file: no_leading_underscores_for_library_prefixes
import 'package:get_it/get_it.dart' as _i174;
import 'package:injectable/injectable.dart' as _i526;
import 'package:terms_bloc_example/data/terms_repository.dart' as _i967;
import 'package:terms_bloc_example/pages/terms_list/bloc/terms_bloc.dart'
  as _i68;

extension GetItInjectableX on _i174.GetIt {
  // initializes the registration of main-scope dependencies inside of GetIt
  _i174.GetIt init({
    String? environment,
    _i526.EnvironmentFilter? environmentFilter,
  }) {
    final gh = _i526.GetItHelper(this, environment, environmentFilter);
    gh.lazySingleton<_i967.TermsRepository>(() => _i967.TermsRepository());
  }
}

```

```

gh.factory<_i68.TermsBloc>(
  () => _i68.TermsBloc(gh<_i967.TermsRepository>()),
);
return this;
}
}

```

5) Инициализация DI и использование в UI в main.dart

```

import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:terms_bloc_example/pages/terms_list/bloc/terms_bloc.dart';
import 'pages/start_page.dart';
import 'pages/terms_list/terms_list_page.dart';
import 'pages/term_detail_page.dart';
import 'routes.dart';
import 'di.dart';

void main() {
  configureDependencies();
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'IT Термины (Bloc)',
      debugShowCheckedModeBanner: false,
      initialRoute: AppRoutes.start,
      routes: {
        AppRoutes.start: (context) => const StartPage(),
        AppRoutes.list: (context) => BlocProvider(
          create: (_) => getIt<TermsBloc>()..add(LoadTermsEvent()),
          child: const TermsListPage(),
        ),
        AppRoutes.detail: (context) => const TermDetailPage(),
      },
    );
  }
}

```

3. Локальное хранение данных на устройстве

При разработке любого приложения необходимо решить, какие данные хранить, как подготовить их структуру и как обеспечить быстрый и надёжный доступ к этой информации.

Во Flutter существуют различные подходы и библиотеки для обеспечения хранения данных приложения и у каждого из них есть свои плюсы и минусы:

1) Хранение файлов с помощью библиотек **path_provider** и **flutter_cache_manager**.

Библиотека **path_provider** используется для получения корректных путей файлов и директорий на устройстве. Каждая операционная система может иметь разные подходы к сохранению данных, а **path_provider** помогает нивелировать эти различия. Класс **File** из библиотек **dart:io** предоставляет асинхронные методы для взаимодействия с файлами. Библиотеки **path_provider** и **dart:io** недоступны на веб-платформе. Это связано с недоступностью файловой системы из веба по соображениям безопасности. В таком случае вместо указанных библиотек можно воспользоваться библиотекой **file_picker**.

Библиотека **flutter_cache_manager** предназначена для кэширования файлов приложения. Это полезный инструмент для управления кэшем и ускорения загрузки файлов.

Особенности и возможности:

- Поддерживает in-memory хранилище для быстрого доступа к кэшированным файлам.
- Кэширование происходит в файловой системе устройства.
- Встроенный механизм управления очередью загрузки.
- Для загрузки файлов использует библиотеку HTTP.

- Часто используется вместе с библиотекой `cached_network_image` для управления кэшированием изображений.
- Доступна на всех платформах.

2) Библиотеки для хранения данных в формате ключ — значение (Key — Value)

Эти библиотеки не являются подвидами каких-либо баз данных (SQL/NoSQL) и представляют собой легковесное и простое решение для специфических задач, таких как кэширование и хранение настроек:

- `shared_preferences`;
- `flutter_secure_storage`;
- `get_storage`.

SharedPreferences предоставляет удобный способ хранения настроек приложения и иных простых данных — сохраняется/читается обыкновенный текстовый файл на iOS в формате `.plist` (`NSUserDefaults`) и на Android в формате `.xml` (`SharedPreferences`), используется для хранения: настроек (например тем: `dark/light`), флажков («Показывать подсказку при старте?»).

flutter_secure_storage — это библиотека для безопасного хранения конфиденциальных данных в приложениях Flutter. Она поддерживает все платформы, что делает её отличным выбором для хранения и управления чувствительными данными. Для более продвинутого хранения данных и их обмена между разными приложениями одного разработчика на iOS существует технология `App Group`, которую также можно конфигурировать для использования с `flutter_secure_storage`. Библиотека отлично подходит для приложений, требующих безопасного хранения таких данных, как токены аутентификации, пароли и другие чувствительные данные.

get_storage — библиотека для Flutter, предназначенная для хранения простых данных в памяти устройства. Она предоставляет простой и интуитивно понятный интерфейс для работы с хранилищем и используется для хранения настроек пользователя, временных данных и другой информации, которая не требует сложных манипуляций с базами данных.

3) Библиотеки для работы с базами данных

а) Реляционные (SQL) базы данных подходят для хранения структурированных данных (например хэширование товаров в онлайн-магазине, т.к. структура данных включает в себя сложные связи, такие как отношения товара с категорией и другими параметрами): `SQLite`, `Drift`

б) Нереляционные (NoSQL) базы данных: `Hive`, `Isar`, `ObjectDB` и т.д.

Реляционные (SQL) базы данных

SQLite — это плагин для Flutter, который представляет собой обёртку над нативными реализациями `SQLite`, встраиваемой реляционной базы данных. Он предоставляет мощные средства для хранения и управления данными в приложении, такие как SQL-запросы, транзакции, миграции.

Особенности и возможности:

- Хранение сложно структурированных данных — подходит для реляционных баз данных.
- Использует платформенные реализации `SQLite` для Android и iOS.
- Не поддерживается в веб-версии Flutter.
- Позволяет выполнять SQL-запросы для выборки и модификации данных.
- Данные хранятся в одном файле, что облегчает управление.

Drift — мощная библиотека для хранения данных (ранее известная как `Moor`), тоже использующая «под капотом» `SQLite`. Среди сильных сторон этой библиотеки стоит отметить поддержку всех платформ, миграций, тонких настроек открытия файла БД.

Нереляционные (NoSQL) базы данных

Hive — это очень быстрое NoSQL-хранилище, предназначенное для сохранения данных в формате «ключ — значение» (`key-value`). `Hive` позволяет вам сохранять не только примитивные данные, но и сложные объекты. Он предоставляет унифицированный способ хранения данных на всех поддерживаемых платформах (`mobile/desktop/web`).

Центральное понятие в `Hive` — это сущность `Box`. Она используется для организации и хранения данных. Каждый `Box` представляет собой файл, созданный в рабочей директории.

Одно приложение может содержать несколько `Box`, и их можно рассматривать как таблицы в реляционных базах данных. При открытии содержимое `Box` полностью считывается и загружается в кэш, что обеспечивает быстрый доступ.

Isar также использует кодогенерацию для создания таблиц и обеспечивает множество стандартных удобных возможностей для работы с данными, таких как ACID-транзакции (обеспечивают надёжное и безопасное хранение данных), производительность операций с данными (библиотека написана на Rust).

Работа с SQL базой данных Drift

Библиотека Drift представляет собой высокоуровневую обертку над SQLite, которая позволяет описывать таблицы прямо на Dart, генерирует нужные модели и SQL-запросы автоматически, умеет подключаться к базе данных через Stream.

1) В файл pubspec.yaml проекта добавить пакет drift в качестве зависимости:

dependencies:

drift: ^2.29.0

drift_flutter: ^0.2.7

sqlite3_flutter_libs: ^0.5.0

path_provider: ^2.1.5

dev_dependencies:

drift_dev: ^2.29.0

build_runner: ^2.10.4

В качестве альтернативы можно в терминале ввести следующую команду:

```
dart pub add drift drift_flutter sqlite3_flutter_libs path_provider dev:drift_dev dev:build_runner
```

2) Импорт библиотек:

```
import 'package:drift/drift.dart';
```

```
import 'package:drift_flutter/drift_flutter.dart';
```

3) Инициализация базы данных

Создаем файл с конфигурацией базы данных. Т.к. его название никак не регламентировано, это может быть: database.dart, db.dart, drift.dart и т.д. (класс Users (таблица пользователей))

Файл lib/database.dart

```
import 'package:drift/drift.dart';
```

```
import 'package:drift_flutter/drift_flutter.dart';
```

```
import 'package:drift_flutter/user_table.dart';
```

```
/// Путь до файла, в котором будет сохраняться сгенерированный код
```

```
part 'database.g.dart';
```

```
@DriftDatabase(tables: [Users]) // Передаем список таблиц
```

```
class AppDatabase extends _$AppDatabase {
```

```
  AppDatabase() : super(_openConnection());
```

```
  @override
```

```
  int get schemaVersion => 1; /// Текущая версия базы данных
```

```
///Открытие соединения с базой данных
```

```
LazyDatabase _openConnection() {
```

```
  return driftDatabase(
```

```
    name: 'database_name'); // name – имя БД
```

```
    native: const DriftNativeOptions(
```

```
      databaseDirectory: getApplicationSupportDirectory, //возвращает системную папку
```

```
    ),
```

```
  );
```

```
});
```

```
}
```

4) Запуск кодогенерации

В терминале:

```
dart run build_runner build --delete-conflicting-outputs
```

5) Объявление подключения к базе данных

В файле «main.dart» объявляем подключение к БД: если она будет отсутствовать на устройстве, запустится стадия ее создания и инициализации:

```
import 'package:drift_sqlite/database.dart';
import 'package:flutter/material.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  final database = AppDatabase();
  // Передаем базу данных через DI или напрямую в приложение
  runApp(MyApp(database: database)); }
```

Таблицы и схемы

При использовании Drift не нужно вручную писать запросы на создание таблиц: это сделает кодогенератор. Таблицы БД описываются как классы Dart, не прибегая к чистому SQL.

Пример описания таблицы пользователей – Users:

Файл lib/users_table.dart

```
import 'package:drift/drift.dart';

class Users extends Table {
  IntColumn get id => integer().autoIncrement();
  TextColumn get name => text().nullable();
  TextColumn get email => text().named("email");
  TextColumn get post => text().nullable();
  IntColumn get age => integer();
  DateTimeColumn get createdAt => dateTime().withDefault(currentDateAndTime());
}
```

Операции с данными

а) Вставка данных в таблицу

```
final _db = AppDatabase();
await _db.into(_db.users).insert(
  UsersCompanion( //Users – имя класса с описанием таблицы
    name: const Value('Василий'),
    email: Value('vasily@example.com'),
    post: Value('Manager'),
    age: const Value(30),
  ), );
```

б) Выборка данных

```
final users = await _db.select(_db.users).get();
Выборка данных по условию
// getSingle() вернет не список, а один объект
final user = await (_db.select(_db.users)..where(
  (t) => t.id.equals(1)
)).getSingle();
```

в) Обновление данных

```
await (_db.update(_db.users)..where((t) => t.id.equals(1))).write(
  /// Меняем имя первого пользователя на «Михаил»
  UsersCompanion(
    name: Value('Михаил'),
  ), );
}
```

UsersCompanion – модель-помощник, которая может использоваться при таких операциях, как: insert, update и replace. К тому же она позволяет:

- указать только необходимые для запроса параметры;
- явно указать какое значение присутствует, а какое - нет;
- работать с nullable-полями и autoincrement-id, которые нельзя задать в обычной модели.

г) Удаление данных

```
await (_db.delete(_db.users)..where((t) => t.id.equals(1))).go();
```

Удалить все данные из таблицы

```
await _db.delete(_db.users).go();
```

Миграции

Миграции (migrations) — это инструкции, которые описывают, как изменить структуру базы данных, когда схема обновляется.

Drift отслеживает schemaVersion: когда мы увеличиваем schemaVersion, Drift вызывает:

```
MigrationStrategy get migration => MigrationStrategy(
```

```
  onCreate: (m) => m.createAll(),
```

```
  onUpgrade: (m, from, to) async {
```

```
    // миграции
```

```
  },
```

```
);
```

Структура миграций Drift

onCreate

Вызывается один раз, когда база данных создаётся впервые.

onUpgrade

Вызывается каждый раз при увеличении schemaVersion. Здесь описывается, что делать со старыми данными.

Пример:

```
onUpgrade: (m, from, to) async {
```

```
  if (from == 1) {
```

```
    await m.addColumn(users, users.group);
```

```
  }
```

```
}
```

beforeOpen

Запускается каждый раз, когда база открывается.

Для примера реализации Data Layer изменим наше приложение с использованием SQL базы данных Drift.

Пример мобильного приложения с 3-мя экранами с использованием пакетов: flutter_bloc для управления состоянием, GetIt + Injectable для управления зависимостями, Drift для локального хранения данных на устройстве.

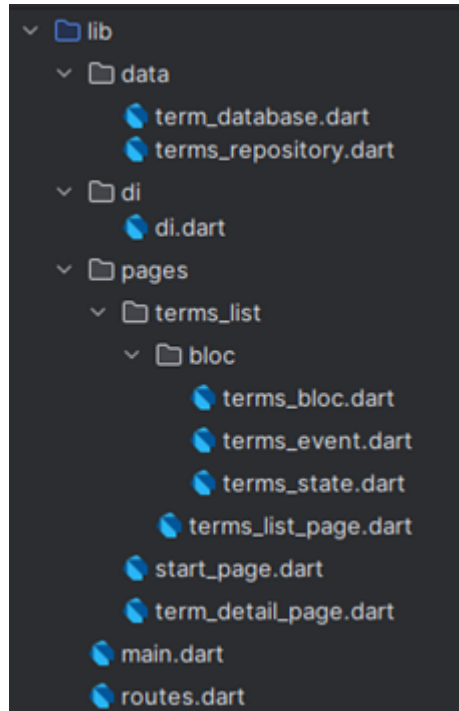
Выполнение:

1) В файл pubspec.yaml проекта добавить необходимые пакеты в качестве зависимостей:

```
dependencies:
  drift_flutter: ^0.2.7
  sqlite3_flutter_libs: ^0.5.0
  path: ^1.9.0
  path_provider: ^2.1.5
  flutter_bloc: ^9.1.1
  get_it: ^9.0.5
  injectable: ^2.6.0

dev_dependencies:
  drift_dev: ^2.29.0
  injectable_generator: ^2.9.1
  build_runner: ^2.10.4
```

2) Создать структуру приложения, добавив необходимые папки и файлы в соответствии с рисунком.



3) Разработать программный код приложения

Файл с конфигурацией базы данных и таблицей (класс Terms): data/term_database.dart для автоматической генерации файла term_database.g.dart

```
import 'package:drift/drift.dart';
import 'package:drift_flutter/drift_flutter.dart';
import 'package:path_provider/path_provider.dart';
import 'package:injectable/injectable.dart';

part 'term_database.g.dart';

/// Таблица терминов
class Terms extends Table {
  IntColumn get id => integer().autoIncrement();
  TextColumn get title => text();
  TextColumn get shortDescription => text();
  TextColumn get fullDescription => text();
}

/// Drift-база
@lazySingleton
@DriftDatabase(tables: [Terms])
class AppDatabase extends _$AppDatabase {
  AppDatabase() : super(_openConnection());

  @override
  int get schemaVersion => 1;

  /// Стратегия миграций и первичное заполнение
  @override
  MigrationStrategy get migration =>
    MigrationStrategy(
      onCreate: (m) async {
        // создаем все таблицы
        await m.createAll();

        // Первоначальное заполнение (как раньше наш List<Term>)
        await batch((
          batch) { //batch - объект типа Batch, который предоставляет удобный способ
            //выполнить несколько SQL-операций за один проход
            batch.insertAll(TermsCompanion.insert(
              title: 'API',
```

```

        shortDescription: 'Интерфейс для взаимодействия программ',
        fullDescription:
        'API (Application Programming Interface) – набор правил и спецификаций, '
        'по которым одна программа может взаимодействовать с другой.',
    ),
    TermsCompanion.insert(
        title: 'HTTP',
        shortDescription: 'Протокол передачи гипертекста',
        fullDescription:
        'HTTP (HyperText Transfer Protocol) – протокол прикладного уровня '
        'для передачи данных в интернете.',
    ),
    TermsCompanion.insert(
        title: 'JSON',
        shortDescription: 'Текстовый формат обмена данными',
        fullDescription:
        'JSON (JavaScript Object Notation) – лёгкий текстовый формат '
        'обмена данными человеко-читаемого вида.',
    ),
  ],
);
},
);
}

/// функция открытия соединения (для Flutter)
LazyDatabase _openConnection() {
  return LazyDatabase(() async {
    return driftDatabase(
      name: 'terms_drift.db',
      native: const DriftNativeOptions(
        databaseDirectory: getApplicationSupportDirectory, //возвращает системную папку
      ),
    );
  });
}

```

4) Выполняем автоматическую генерацию классов введя в терминале команду:

```
flutter pub run build_runner build --delete-conflicting-outputs
```

Появится файл lib/data/term_database.g.dart

5) Файл репозитория (data/term_repository.dart)

```

import 'package:injectable/injectable.dart';
import 'term_database.dart';

@LazySingleton()
class TermsRepository {
  final AppDatabase _db;

  TermsRepository(this._db);

  Future<List<Term>> fetchTerms() async {
    // Term – это дата-класс, сгенерированный Drift'ом по таблице Terms
    return _db.select(_db.terms).get(); //получить список
  }

  /// Получить объект Term по термину (для поиска)
  Future<Term?> getTermByTermin(String termin) async {
    return (_db.select(_db.terms)
      ..where((t) => t.title.equals(termin)))
      .getSingleOrNull();
  }
}

```

6) Файлы BLoC для страницы со списком терминов

Файл с классом событий (pages/terms_list/bloc/terms_event.dart)

```
part of 'terms_bloc.dart';
```

```
@immutable
sealed class TermsEvent {}
class LoadTermsEvent extends TermsEvent {}
```

Файл с классом состояний (pages/terms_list/bloc/terms_state.dart)

```
part of 'terms_bloc.dart';

@immutable
sealed class TermsState {}

final class TermsInitial extends TermsState {}

final class TermsLoading extends TermsState {}

final class TermsLoaded extends TermsState {
  final List<Term> terms;
  TermsLoaded(this.terms);
}

final class TermsError extends TermsState {
  final String message;
  TermsError(this.message);
}
```

Файл с логикой блока (pages/terms_list/bloc/terms_bloc.dart)

```
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';
import 'package:injectable/injectable.dart';

import '../../../data/terms_repository.dart';
import '../../../data/term_database.dart';

part 'terms_event.dart';
part 'terms_state.dart';

@injectable
class TermsBloc extends Bloc<TermsEvent, TermsState> {
  final TermsRepository repository;

  TermsBloc(this.repository) : super(TermsInitial()) {
    on<LoadTermsEvent>(_onLoadTerms);
  }

  Future<void> _onLoadTerms(
    LoadTermsEvent event,
    Emitter<TermsState> emit,
  ) async {
    emit(TermsLoading());
    try {
      final terms = await repository.fetchTerms();
      emit(TermsLoaded(terms));
    } catch (e) {
      emit(TermsError('Ошибка загрузки данных: $e'));
    }
  }
}
```

7) Файл с DI-контейнером (di/di.dart)

```
import 'package:get_it/get_it.dart';
import 'package:injectable/injectable.dart';
import 'di.config.dart';

final getIt = GetIt.instance;

/// Регистрируем все зависимости
@InjectableInit()
void setupDI() => getIt.init();
```

Выполняем автоматическую генерацию классов, введя в терминале команду:

flutter pub run build_runner build --delete-conflicting-outputs

Появится файл di/di.config.dart

8) Файл с классом маршрутов (routes.dart)

```
• class AppRoutes {  
    static const start = '/';  
    static const list = '/list';  
    static const detail = '/detail';  
}
```

9) Файлы слоя UI

Файл со начальной страницей – кнопка Показать (pages/start_page.dart)

```
import 'package:flutter/material.dart';  
  
import '../routes.dart';  
  
class StartPage extends StatelessWidget {  
    const StartPage({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return Scaffold(  
            appBar: AppBar(title: const Text('Стартовый экран')),  
            body: Center(  
                child: ElevatedButton(  
                    onPressed: () => Navigator.pushReplacementNamed(context, AppRoutes.list),  
                    child: const Text('Показать ИТ-термины'),  
                ),  
            ),  
        );  
    }  
}
```

Файл со страницей списка терминов (pages/term_list/terms_list_page.dart)

```
import 'package:flutter/material.dart';  
import 'package:flutter_bloc/flutter_bloc.dart';  
  
import 'package:drift_example/pages/terms_list/bloc/terms_bloc.dart';  
import '../../../di/di.dart';  
import '../../../routes.dart';  
  
class TermsListPage extends StatelessWidget {  
    const TermsListPage({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return BlocProvider(  
            create: (_) => getIt<TermsBloc>()..add(LoadTermsEvent()),  
            child: Scaffold(  
                appBar: AppBar(title: const Text('Список ИТ-терминов')),  
                body: BlocBuilder<TermsBloc, TermsState>(  
                    builder: (context, state) {  
                        if (state is TermsLoading) {  
                            return const Center(child: CircularProgressIndicator());  
                        } else if (state is TermsError) {  
                            return Center(child: Text(state.message));  
                        } else if (state is TermsLoaded) {  
                            final terms = state.terms;  
                            if (terms.isEmpty) {  
                                return const Center(child: Text('Список пуст'));  
                            }  
                        }  
  
                        return ListView.builder(  
                            itemCount: terms.length,  
                            itemBuilder: (context, index) {  
                                final term = terms[index];  
                                return Card(  
                                    margin:  

```

```

        const EdgeInsets.symmetric(horizontal: 12, vertical: 6),
        child: ListTile(
          title: Text(term.title),
          subtitle: Text(term.shortDescription),
          trailing: const Icon(Icons.chevron_right),
          onTap: () {
            Navigator.pushNamed(
              context,
              AppRoutes.detail,
              arguments: term,
            );
          },
        ),
      ),
    );
  },
);
}

return const Center(child: Text('Загрузка данных...'));
},
),
),
);
}
}
}

```

Файл со страницей детализации термина (pages/term_detail_page.dart)

```

import 'package:flutter/material.dart';
import '../data/term_database.dart'; // тип Term

class TermDetailPage extends StatelessWidget {
  const TermDetailPage({super.key});

  @override
  Widget build(BuildContext context) {
    final term = ModalRoute.of(context)!.settings.arguments as Term;

    return Scaffold(
      appBar: AppBar(title: Text(term.title)),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Text(
          term.fullDescription,
          style: const TextStyle(fontSize: 16),
        ),
      ),
    );
  }
}

```

10) Файл main.dart

```

import 'package:flutter/material.dart';

import 'di/di.dart';
import 'pages/start_page.dart';
import 'pages/terms_list/terms_list_page.dart';
import 'pages/term_detail_page.dart';
import 'routes.dart';

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  setupDI();
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
}

```

```

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'IT Термины (Drift)',
    debugShowCheckedModeBanner: false,
    initialRoute: AppRoutes.start,
    routes: {
      AppRoutes.start: (_) => const StartPage(),
      AppRoutes.list: (_) => const TermsListPage(),
      AppRoutes.detail: (_) => const TermDetailPage(),
    },
  );
}

```

4. Задание на практическую работу

Изменить приложение, разработанное в предыдущей практической работе, выполнив следующие действия:

- 1) Заменить тестовые данные на полноценный слой данных, реализованный на базе локальной SQLite-БД через библиотеку Drift (в БД должна храниться информация о пунктах/элементах списков, включая изображения).
- 2) Реализовать паттерн Repository, содержащий необходимые CRUD-операции (минимально – получение информации обо всех пунктах/элементах списков).
- 3) Настроить DI-контейнер (GetIt + Injectable), обеспечив автоматическую регистрацию и внедрение зависимостей: экземпляра базы данных Drift, репозитория и соответствующего BLoC. Реализация должна исключать ручное создание этих объектов в UI-слое.
- 4) В отчете представить:
 - цель работы;
 - скриншот структуры проекта;
 - описание архитектуры слоя данных, включающее:
 - ✓ структуру таблицы Drift;
 - ✓ логику миграции и начальное заполнение БД;
 - ✓ принцип работы паттерна Repository;
 - ✓ настройку и использование DI-контейнера;
 - скриншоты экранов HomePage и DetailPage после интеграции с Drift;
 - программный код приложения;
 - выводы о том, какие преимущества дает использование БД Drift и DI-контейнера в архитектуре Flutter-приложения.