

# Методические указания к Практической работе №7

## по дисциплине «Разработка кроссплатформенных мобильных приложений»

**Тема работы: Разработка архитектуры Flutter-приложения (без слоя данных) с использованием архитектурного паттерна BLoC**

**Цель работы:** Сформировать навыки проектирования архитектуры Flutter-приложения с разделением кода на слои и использованием архитектурного паттерна BLoC для управления состоянием экранов.

**План практической работы:**

- Архитектура приложения. Архитектурные паттерны. Структура проекта.
- Стейт-менеджмент (State Management). Архитектурный паттерн BLoC.
- Выполнение практической работы №7 в соответствии с заданием

**Последовательность выполнения практической работы:**

### 1. Архитектура приложения. Архитектурные паттерны. Структура проекта.

Архитектура приложения определяет структуру, взаимосвязь между компонентами и их взаимодействия. Четко разработанная архитектура не только упрощает процесс разработки, но и обеспечивает масштабируемость, гибкость и удобство сопровождения кода.

Архитектура приложения — это не просто распределение файлов по папкам, а система принципов организации кода.

У Flutter, как и у любой другой технологии, есть свои особенности, которые влияют на выбор архитектурных решений. Реактивный подход к обновлению UI, устройство фреймворка, структуры виджетов и их обновления — всё это создаёт уникальный контекст для архитектурных решений.

**Основные архитектурные концепции, применимые к Flutter-разработке:**

- Разбиение на слои.
- Организация файловой структуры проекта: «сперва слои»/«сперва фичи».
- Управление состоянием в приложении (State Management).
- Управление зависимостями (Dependency Injection (DI))

**Архитектура Flutter-приложения** — это логическая организация кода, определяющая:

- как данные хранятся и передаются между компонентами;
- где выполняется бизнес-логика;
- как отделяются интерфейс, логика и данные.

В соответствии с одним из главных принципов Clean Architecture программный продукт разделяется на слои, каждый из которых выполняет конкретную функцию и не зависит от другого. Слои взаимодействуют между собой через чётко определённые интерфейсы.

Таким образом, 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.

#### Logic Layer (Доменный слой)

Реализует бизнес-логику, содержит логику преобразования данных для UI, поступающих с уровня данных.

#### Data Layer (Слой данных)

Взаимодействует с внешними источниками данных, включая базы данных, API и сервисы (управляет доступом к данным, получая их из локальных (например: Hive, SQLite, Drift, SharedPreferences) или удалённых (например: Firebase, PostgreSQL) источников).

Дополнительно может выделяться **App Layer (Слой приложения)**, отвечающий за конфигурацию всего приложения (настройка роутинга, тем, локализации, DI, инициализацию сервисов, главная точка входа (main.dart)).

## Архитектурные паттерны

Для упрощения разработки и стандартизации структуры кода используют архитектурные паттерны.

Архитектурные шаблоны обеспечивают структурированный подход к проектированию программного обеспечения. Они помогают разработчикам организовать код таким образом, чтобы он был удобным для поддержки, масштабирования и тестирования. Во Flutter архитектурные шаблоны в первую очередь фокусируются на управлении состоянием и разделении задач.

### MVP (Model-View-Presenter)

Паттерн Model-View-Presenter (MVP) позволяет разделить обязанности между компонентами приложения. В основе его концепции лежит разделение кода на три основные части: Model, View и Presenter, каждая из которых имеет свои четкие обязанности:

- View. Отвечает за отображение данных и взаимодействие с пользователем. Не содержит бизнес-логики. Отправляет событие в Presenter для изменения Model.
- Presenter – посредник между Model и View. Получает данные из Model и передает их View для отображения. Обрабатывает события пользовательского интерфейса и обновляет Model.
- Model. Отвечает за управление данными, включая бизнес-логику и взаимодействие с источниками данных. Не зависит от пользовательского интерфейса.

### MVC (Model-View-Controller)

Паттерн Model-View-Controller (MVC), как и рассмотренный ранее MVP, позволяет разделить обязанности между компонентами приложения. Отличие же заключается в том, что вся работа шаблона стартует с компонента Controller, который принимает события от пользователя и отправляет запрос на изменения данных в Model. Тот в свою очередь уведомляет View (пользовательский интерфейс), что данные поменялись, запуская тем самым процесс обновления состояния интерфейса пользователя.

### MVVM (Model-View-ViewModel)

Паттерн MVVM обеспечивает разделение интерфейса, бизнес-логики и данных.

- Model: представляет уровень данных приложения, такой как сетевые запросы, базы данных и API.
- View: компонент пользовательского интерфейса, который отслеживает изменения в ViewModel и соответствующим образом обновляет пользовательский интерфейс.
- ViewModel: выступает в качестве посредника между View и Model, храня логику приложения и управляя потоком данных между ними.

### BLoC (Business Logic Component)

BLoC строится на потоках (Streams) и реактивном программировании, чётко разделяя события и состояния. События поступают в BLoC, преобразуются в новые состояния, а интерфейс автоматически реагирует на эти изменения.

У каждого шаблона есть свои сильные стороны и варианты использования, поэтому важно выбрать тот, который лучше всего соответствует требованиям вашего приложения.

## Структура проекта

По принципу разделения ответственности приложение должно быть разделено на слои. Каждый из слоёв, в свою очередь, делится на отдельные компоненты, и каждый компонент:

- имеет свою собственную зону ответственности;
- обладает чётко определённым интерфейсом;
- имеет границы и зависимости, ограниченные своей задачей.

Существует два основных подхода к структурированию кода:

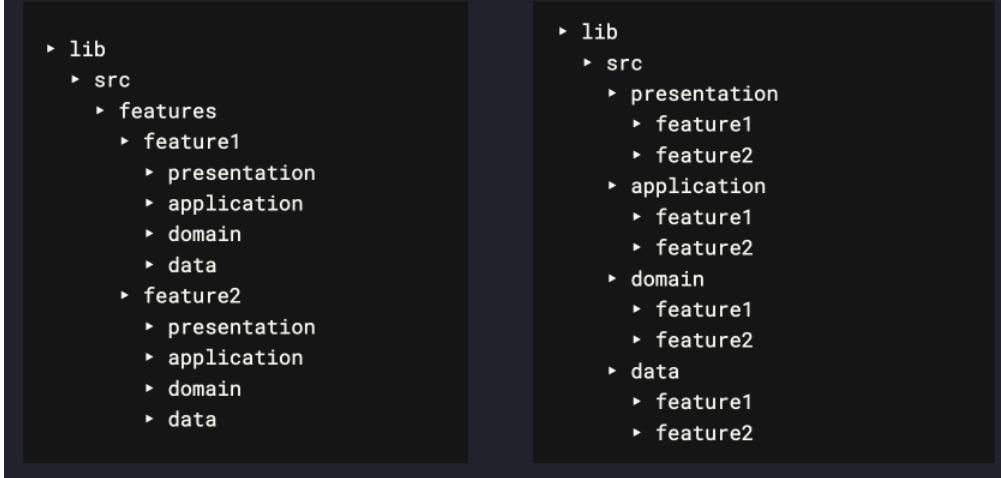
### 1. По функциональности (By feature/Feature-first)

Код группируется по функциональным модулям (фичам), например, auth, profile, detail. Каждый модуль содержит все свои части для этой фичи: UI, логику, модели, репозитории. Архитектура приложений Flutter на основе функций способствует разделению обязанностей, повторному использованию, обслуживаемости и масштабируемости, что делает ее эффективным выбором для разработки сложных и постоянно меняющихся приложений. Каждая функция является автономной и может разрабатываться, тестироваться и обслуживаться независимо.

### 2. По типу (By type/layer-first)

Файлы группируются по архитектурным типам или слоям (например, data, domain, ui или presentation), а в каждом слое уже идут подпапки или файлы, относящиеся к конкретным функциям.

# Feature-first vs Layer-first



## 2. Стейт-менеджмент (State Management). Архитектурный паттерн BLoC.

**Состояние** — это данные, которые влияют на то, как выглядит интерфейс в данный момент.

Примеры состояния:

- пользователь авторизован или нет;
- загружаются ли данные;
- выбрана вкладка или элемент списка;
- результат запроса к серверу.

Когда состояние изменяется — UI должен обновиться.

Во Flutter (как реактивного фреймворка) для этого и нужен State Management (управление состоянием).

**Суть управления состоянием** заключается в эффективной координации данных и пользовательского интерфейса, чтобы любые изменения в данных мгновенно и корректно отражались в UI, и наоборот.

Flutter-приложения требуют организованного подхода к состоянию, чтобы разделять логику и UI, обеспечивая чистоту кода и удобство сопровождения.

Для управления состоянием во Flutter-приложениях используют различные пакеты/подходы:

- setState() для управления состоянием на уровне виджетов (простое обновление состояния внутри StatelessWidget);
- InheritedWidget для передачи данных дочерним виджетам по дереву виджетов без повторного объявления (для общих данных: тем, языков, размеров экрана);
- BLoC (Business Logic Component) подходит для крупных проектов с комплексной логикой, где важно строгое разделение слоёв и возможность модульного тестирования.
- Provider: официальный пакет Flutter, основанный на ChangeNotifier и InheritedWidget, использует концепцию инъекции зависимостей, предоставляя данные там, где они необходимы в дереве виджетов, и перестраивая только те виджеты, которым важна эта часть данных. Он идеально подходит для простых и средних приложений, где важна скорость разработки и ясность архитектуры.

– Riverpod появился как эволюция Provider и устранил привязку провайдеров к контексту виджетов, повышая безопасность кода на этапе компиляции, хорошо масштабируется и позволяет организовать сложные зависимости между блоками состояния без громоздких шаблонных решений.

– Redux: библиотека управления предсказуемыми состояниями, реализующая идею одностороннего потока данных. Он управляет состоянием приложения в едином неизменяемом дереве состояний, которое обновляется путем диспетчеризации действий редукторам, создающим новое состояние, используется в крупных проектах, где важны предсказуемое управление состоянием, история изменений и строгая структура.

– GetX характеризуется минимумом шаблонов, использованием реактивных переменных, встроенным роутингом и DI.

- MobX характеризуют реактивные наблюдаемые переменные; автоматическое обновление UI, подходит для средних проектов с реактивной логикой.

Каждая модель State Management имеет свои плюсы и минусы, которые следует учитывать и соотносить с условиями каждого отдельно взятого проекта.

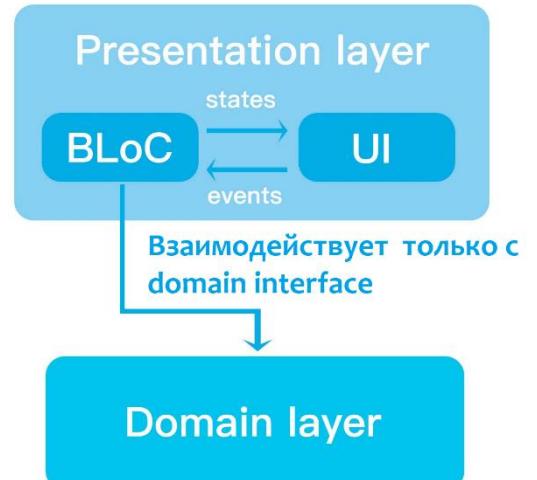
### Паттерн BLoC

Паттерн BLoC — это акроним от «Business Logic Component» (компонент бизнес-логики). Это класс, отделяющий бизнес-логику приложения от пользовательского интерфейса. Такой компонент содержит код, который можно повторно использовать где угодно: в другом модуле, на другой платформе, в другом приложении.

BLoC предлагает структурированный подход к отделению бизнес-логики приложения от его слоя пользовательского интерфейса. В основе BLoC лежит концепция «потоков» (Streams) и принципы реактивного программирования, где входящие «события» (Events) преобразуются в исходящие «состояния» (States).

BLoC — это часть Presentation Layer (слоя представления), но он содержит логику представления и взаимодействует с Domain Layer.

Он идеален для крупномасштабных или корпоративных приложений, где предсказуемые и тестируемые переходы состояния имеют решающее значение.



К основным элементам данного шаблона проектирования относятся:

- Events (События): это входные данные или действия, которые происходят в приложении. Они могут быть инициированы пользователем – пользовательские события (например, нажатие кнопки, отправка формы, ввод текста, пролистывание экрана) или системой – системные события (например, получение данных из сети, сработал таймер, пришло push-уведомление, изменилась ориентация экрана, потеря Интернет-соединения).

Основная цель событий — уведомить BLoC о том, что что-то произошло и требуется изменение состояния приложения.

- State (Состояние): текущее состояние всего графического пользовательского интерфейса приложения или его части, которое меняется в ответ на события (ошибка, ожидание получения данных и т.д.);
- BLoC (бизнес-логика): компонент, принимающий на свой вход события для их последующей обработки и генерации нового состояния. В своей работе использует потоки (Stream).

### Принцип работы BLoC:

1. Пользователь взаимодействует с UI (User Interface), например, нажимает на выключенный переключатель (switch).
2. UI отправляет это событие в соответствующий BLoC.
3. BLoC обрабатывает событие (возможно, обращаясь к Data слою для получения или обновления данных) и генерирует новое состояние в зависимости от ответа и заложенной логики.
4. Новое состояние передается обратно в UI.
5. В зависимости от нового состояния происходит перестроение UI.

### Компоненты паттерна BLoC:

- Stream — это асинхронный поток данных или источник асинхронных событий, доставляемых последовательно.
- Sink — интерфейс, который позволяет нам добавлять данные в Stream.
- StreamController — обёртка, которая предоставляет нам stream и sink и методы управления ими.
- StreamSubscription — подписка на события, которые приходят из Stream.

Изначальная суть паттерна BLoC заключалась в концепции «чёрного ящика» с двумя типами интерфейсов: входы (sinks) и выходы (streams).

Входы принимали данные из разных источников – клики пользователя, ответы API, настройки приложения, а выходы транслировали результаты наружу – обновлённое состояние UI, ошибки или данные для других сервисов.

Внутренняя реализация не регламентировалась: разработчик сам решал, как преобразовывать входы в выходы, будь то асинхронные запросы, комбинация данных или даже игнорирование некоторых событий. Например, BLoC мог агрегировать данные из нескольких входов (поисковый запрос + геолокация) и выдавать через выходы отфильтрованные результаты и статус загрузки.

Со временем паттерн адаптировали под нужды разработчиков. Феликс Анжелов (англ. Felix Angelov), создатель пакетов bloc и flutter\_bloc, предложил упрощённую версию BLoC, где компонент обрабатывал события (events) и выдавал состояния (states) по принципу «один вход — один выход». Это повысило предсказуемость и упростило тестирование, но сузило изначальную концепцию: исчезла поддержка множественных входов/выходов, а бизнес-логика стала сводиться к преобразованию событий в состояния. Позже появился Cubit – ещё более минималистичный вариант, где вместо событий использовались методы.

## Bloc/Cubit

У BLoC существует 2 вида реализации: Bloc и Cubit.

Bloc работает на основе событий и состояний, в то время как Cubit, более простая версия Bloc, работает непосредственно с изменениями состояния.



## Библиотеки bloc/flutter\_bloc

**bloc:** Это базовый пакет, который определяет сам паттерн BLoC и Cubit. Он не зависит от Flutter и содержит основную логику управления состоянием и событиями без привязки к Flutter.

Базовые классы:

`Bloc<Event, State>` — основной класс для описания логики;

`Cubit<State>` — упрощённая версия блока без событий;

`Transition` — объект, описывающий переход от старого состояния к новому.

Механизмы логирования, наблюдения (BlocObserver), тестирования и трансформации потоков.

Используется в проектах на Dart без Flutter (например, серверные, консольные).

**flutter\_bloc:** Этот пакет предоставляет набор виджетов и утилит, специально разработанных для интеграции паттерна BLoC с пользовательским интерфейсом Flutter. Он содержит такие важные компоненты, как BlocProvider, BlocBuilder, BlocListener, BlocConsumer. Эти виджеты позволяют эффективно предоставлять экземпляры BLoC в дерево виджетов, перестраивать UI в ответ на изменения состояния и обрабатывать побочные эффекты.

**BlocProvider** – это фундаментальный виджет для внедрения зависимостей (Dependency Injection), который делает экземпляр BLoC (или Cubit) доступным для всех дочерних виджетов в определенном поддереве. Он гарантирует, что все виджеты в этом поддереве могут получить доступ к одному и тому же экземпляру BLoC, обеспечивая согласованность состояния.

## Параметры BlocProvider

**create:** Этот параметр используется для создания нового экземпляра BLoC.

Когда BlocProvider создается с помощью create, он автоматически управляет жизненным циклом этого BLoC, закрывая его (dispose()) при удалении BlocProvider из дерева виджетов. Это наиболее распространенный и рекомендуемый способ использования BlocProvider для создания и управления BLoC.

`BlocProvider(`

`create: (BuildContext context) => CounterCubit(), // Создает новый Cubit`

`child: CounterPage(),`

`);`

**value:** Этот параметр используется для предоставления уже существующего экземпляра BLoC. В этом случае BlocProvider не управляет жизненным циклом BLoC, и разработчик должен вручную позаботиться о его закрытии (close()), если это необходимо. BlocProvider.value часто используется при навигации на новый экран, где BLoC уже существует в предыдущем контексте и его нужно просто передать дальше.

```
BlocProvider.value(  
    value: BlocProvider.of<CounterCubit>(context), // Использует существующий Cubit  
    child: AnotherPage(),  
,
```

### **MultiBlocProvider**

Когда в одном месте приложения необходимо предоставить несколько разных BLoC, использование вложенных BlocProvider может привести к глубокой и трудночитаемой структуре. MultiBlocProvider позволяет объединить несколько BlocProvider в один, значительно улучшая читаемость кода и избегая такой вложенности.

```
MultiBlocProvider(  
    providers:,  
    child: HomePage(),  
,
```

**BlocBuilder** – это виджет, который требует экземпляр BLoC (или Cubit) и функцию builder. Он отвечает за перестроение части пользовательского интерфейса в ответ на новые состояния, излучаемые BLoC. По своей сути он очень похож на StreamBuilder, но предлагает более простой API, специально адаптированный для BLoC. Функция builder должна быть «чистой» функцией, которая возвращает виджет в ответ на текущее состояние.

BlocBuilder используется, когда необходимо отобразить различные виджеты или обновить UI в зависимости от текущего состояния BLoC. Например, он может использоваться для показа индикатора загрузки, отображения списка данных после их получения или вывода сообщения об ошибке, если что-то пошло не так.

Параметр buildWhen виджета BlocBuilder позволяет тонко контролировать, когда функция builder должна быть вызвана. buildWhen принимает предыдущее и текущее состояния BLoC и возвращает true, если виджет должен перестроиться, и false в противном случае. Это мощный инструмент для оптимизации производительности, предотвращающий ненужные перестроения UI.

**BlocListener** используется для выполнения действий, которые должны произойти один раз в ответ на изменение состояния BLoC, но не приводят к перестроению UI. Это идеальный выбор для обработки «побочных эффектов» (side effects), таких как навигация между экранами, отображение Snackbar, Dialog сообщений, а также для вызова функций, не связанных с рендерингом.

Параметр **listenWhen**, подобно buildWhen, позволяет контролировать, когда функция listener должна быть вызвана. Это предотвращает вызов слушателя при каждом изменении состояния, если это не требуется, и обеспечивает точное срабатывание побочных эффектов.

**BlocConsumer** – это удобный виджет, который объединяет функциональность BlocListener и BlocBuilder, значительно уменьшая бойлерплейт, который мог бы возникнуть при их использовании по отдельности. Он предоставляет как функцию listener для обработки побочных эффектов, так и функцию builder для перестройки UI. BlocConsumer следует использовать, когда вам нужно одновременно перестраивать UI и выполнять побочные эффекты в ответ на изменения состояния одного и того же BLoC.

```
BlocConsumer<AuthBloc, AuthState>(  
    // Условие для вызова listener: только при ошибке аутентификации  
    listenWhen: (previousState, currentState) => currentState is AuthError,  
    listener: (context, state) {  
        if (state is AuthError) {  
            ScaffoldMessenger.of(context).showSnackBar(  
                SnackBar(content: Text('Ошибка входа: ${state.message}')),  
            );  
        }  
    },
```

```

// Условие для вызова builder: при загрузке или успешной аутентификации
buildWhen: (previousState, currentState) => currentState is AuthLoading || currentState is AuthSuccess,
builder: (context, state) {
  if (state is AuthLoading) {
    return const Center(child: CircularProgressIndicator()); // Показываем индикатор загрузки
  } else if (state is AuthSuccess) {
    return const Center(child: Text('Добро пожаловать!')); // Показываем приветствие
  }
  return const Center(child: Text('Пожалуйста, войдите.')); // Дефолтный UI
}, );

```

### Extension-методы пакета flutter\_bloc для работы с Bloc и Cubit

Метод	Что делает	Когда использовать
<code>context.read&lt;T&gt;()</code>	Получает экземпляр блока (или кубита) один раз и не слушает изменения состояния	Когда нужно вызвать метод или добавить событие (add, emit), но не нужно перестраивать UI
<code>context.watch&lt;T&gt;()</code>	Подписывается на изменения состояния блока и перестраивает виджет при каждом обновлении	Когда нужно, чтобы виджет обновлялся при изменении состояния (альтернатива BlocBuilder)
<code>context.select&lt;T,R&gt;(Bloc&lt;T&gt; bloc =&gt; R)</code>	Подписывается только на конкретное поле состояния, а не на весь объект	Когда нужно отслеживать только часть состояния (например, один параметр)
<code>context.listen&lt;T&gt;()</code>	Реагирует на изменение состояния (вызывает callback), но не перестраивает UI	Для побочных эффектов: показ Snackbar, переход на другой экран, диалоговые окна

**Алгоритм работы при создании новых экранов с использованием архитектуры BLoC (пакет flutter\_bloc):**

**1) Определить события (Events) и состояния (States) для нового экрана**

События описывают действия пользователя или системные сигналы (например, нажатие кнопки, получение данных), а состояния — данные, которые должен отображать интерфейс.

**2) Создать Bloc-класс, унаследованный от Bloc<Event, State>**

В конструкторе определить начальное состояние через super(initialState) и зарегистрировать обработчики событий с помощью метода on<Event>().

**3) Реализовать бизнес-логику внутри обработчиков событий**

При необходимости запрашивать данные из репозиториев или сервисов, обрабатывать их и передавать новое состояние через emit(newState).

**4) Создать новый экран (виджет), который будет отображать данные и реагировать на изменения состояния**

**5) Обернуть экран в BlocProvider (или MultiBlocProvider) и передать в него экземпляр созданного блока:**

Пример

BlocProvider(

  create: (\_) => AuthBloc(),

  child: AuthScreen(),

);

**6) Использовать BlocBuilder, BlocListener или BlocConsumer для подписки на состояние блока:**

BlocBuilder — перестраивает UI при изменении состояния

BlocListener — выполняет одноразовые действия (например, показывает Snackbar)

BlocConsumer — объединяет обе возможности

**7) Отправлять события в блок из UI через вызов:**

`context.read<AuthBloc>().add(LoginButtonPressed());`

## Использование пакета flutter\_bloc

1) Файл pubspec.yaml

dependencies:

  flutter\_bloc: ^9.1.1

2) В файле dart

```
import 'package:flutter_bloc/fkutter_bloc.dart';
```

## Автоматическая генерация BLoC/Cubit через расширение (плагин) bloc

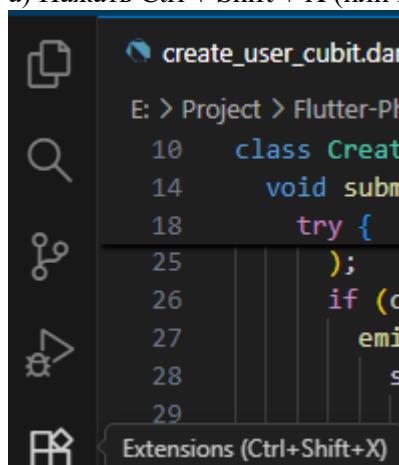
Расширение bloc (доступно как для VSCode, так и Android Studio):

1. Автоматически генерировать BLoC/Cubit классы в соответствующих файлах (название\_event.dart для Bloc, название\_state.dart, название\_bloc.dart/название\_cubit.dart).

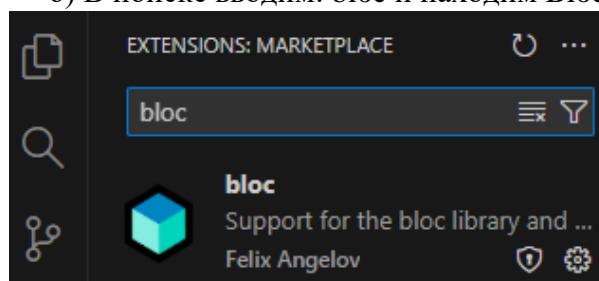
### VS Code

1) Установка расширения/плагина

а) Нажать Ctrl + Shift + X (или клик по иконке Extensions).



б) В поиске вводим: bloc и находим Bloc (Автор: Felix Angelov)



в) Устанавливаем и restart IDE

2) Использование в VS Code

а) Открыть палитру команд (Ctrl + Shift + P) и ввести: Bloc: New bloc / Bloc: New Cubit

б) Далее ввести имя (например, auth) и папку (например, bloc)

После этого после этого будет создана структура со стандартными шаблонами, например:  
auth/

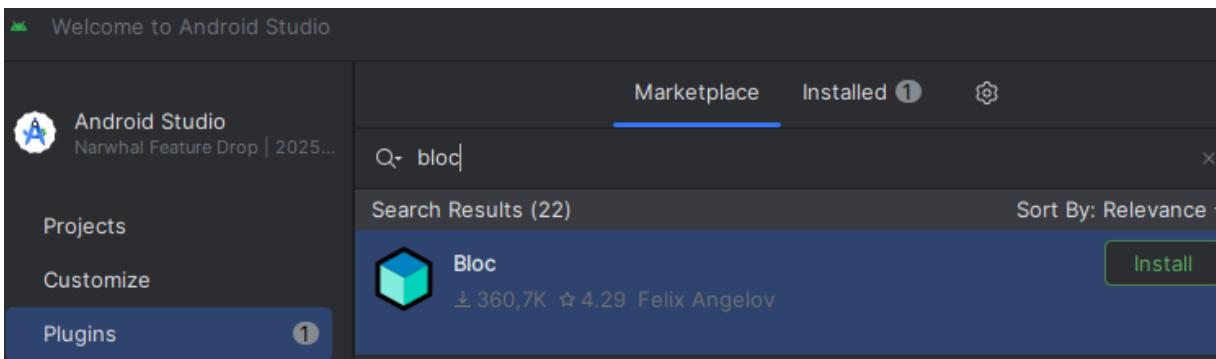
```
└── bloc/
    ├── auth_bloc.dart
    ├── auth_event.dart
    └── auth_state.dart
```

## Android Studio

1) Установка расширения/плагина:

а) File – Settings – Plugins (на macOS: Android Studio – Preferences – Plugins).

б) В поиске вводим: bloc и находим Bloc (Автор: Felix Angelov)



в) Устанавливаем и restart IDE

## 2) Использование в Android Studio

а) Правая кнопка мыши по папке (например, bloc) – New – Bloc Class или New - Cubit Class

б) В появившемся окне вводим имя (например login): IDE создаст файлы login\_bloc.dart, login\_event.dart, login\_state.dart (BLoC) / login\_bloc.dart, login\_state.dart (Cubit)

2. Расширение предоставляет также **Code Actions/Intention Actions (обертывание виджетов)**, которое позволяет быстро обернуть существующие виджеты в BLoC-специфичные виджеты, такие как: BlocBuilder, BlocListener, BlocConsumer, BlocProvider и т.п.

### VS Code (Code Actions):

- 1) Курсор поставить на виджет, который необходимо обернуть (например, Center).
- 2) Нажать Ctrl + «.» (Windows/Linux) или Cmd + «.» (Mac).
- 3) В появившемся меню выбираешь нужное действие (например: Wrap with BlocBuilder)
- 4) IDE автоматически обворачивает твой виджет в BlocBuilder с готовым шаблоном:  
builder: (context, state) { ... }.

### Android Studio (Intention Actions):

- 1) Навести курсор на нужный виджет (например, Center).
- 2) Нажать Alt + Enter (или Option + Enter на macOS).
- 3) В появившемся меню выбрать действие (например: Wrap with BlocBuilder)
- 4) Выбрать нужное действие: IDE сама обернёт код.

3. Плагин **bloc** поставляется с обширным набором **сниппетов (snippet)** – шаблонов кода, которые позволяют быстро вставить часто используемые конструкции BLoC по короткому ключу. В Android Studio сниппеты часто называются Live Templates.

### VS Code/Android Studio

- 1) В файле Dart (например, auth\_bloc.dart) начинаем печатать ключевое слово, например: bloc
- 2) IDE предложит вариант автозавершения: bloc
- 3) Нажимаем Enter или Tab и плагин автоматически вставит шаблон кода

#### Сниппеты:

bloc: создает базовый класс BLoC

cubit: создает базовый класс Cubit

blocbuilder: создает шаблон BlocBuilder

blocprovider: создает шаблон BlocProvider

read, watch, select: Сниппеты для расширений BuildContext и т.п.

**Пример** приложения с управлением состоянием с помощью паттерна BLoC (пакет flutter\_bloc) без слоя данных.

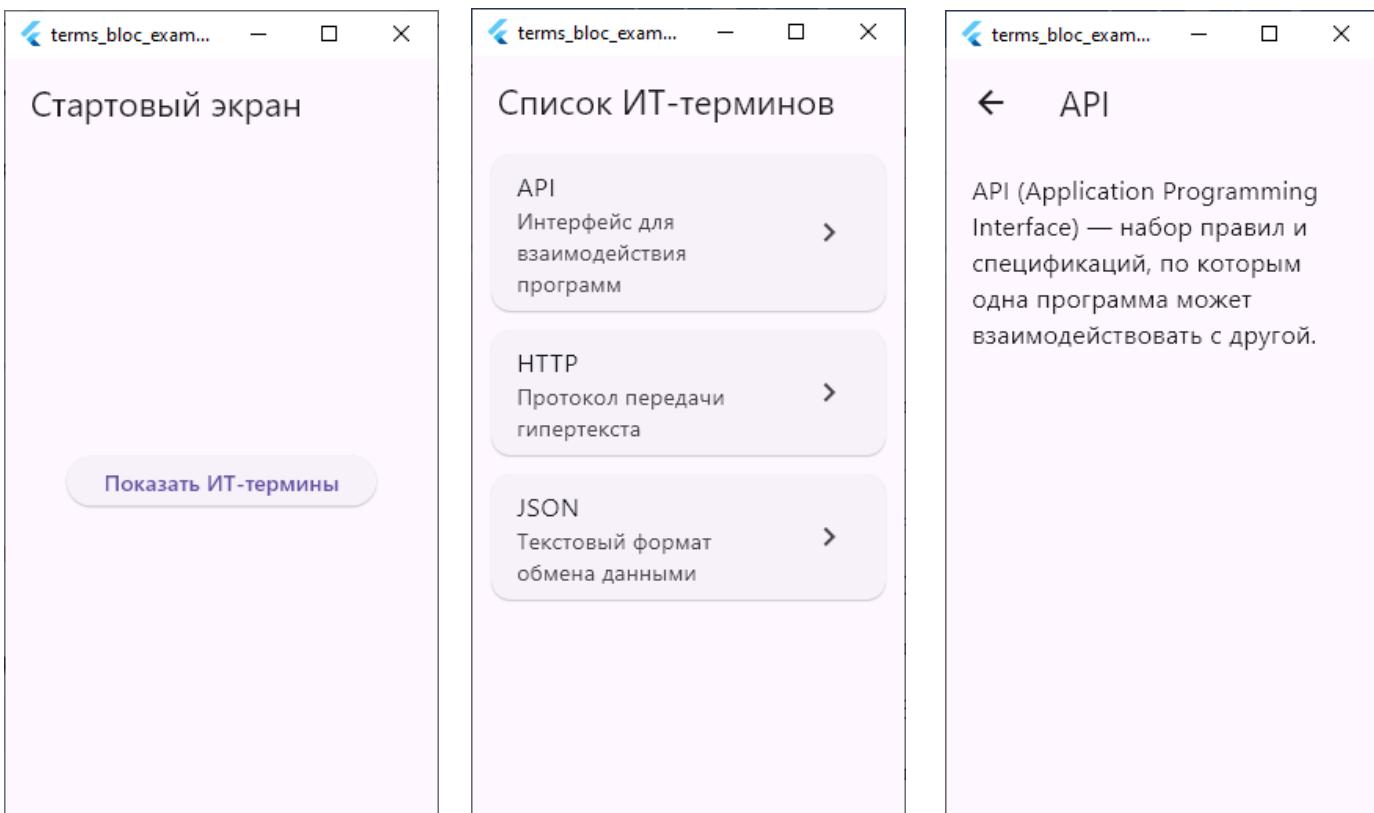
Приложение состоит из 3-х экранов:

- 1) Стартовый экран (кнопка «Показать»)
- 2) Экран списка IT-терминов
- 3) Экран детализации термина

Данные (список терминов) хранится в файле data/terms\_data.dart в виде коллекции List<Term>, где Term – класс для описания терминов.

При загрузке данных (событие для кнопки Показать) отображаются состояния страницы со списком терминов:

- initial — когда у нас ничего нет (экран пустой);
- loading — индикатор загрузки;
- error — сообщение об ошибке;
- loaded — отображение списка терминов.



### Выполнение:

1) В файле pubspec.yaml проекта добавить пакет flutter\_bloc в качестве зависимости:

dependencies:

  flutter\_bloc: ^9.1.1

2) Для генерации шаблонного кода добавить расширение bloc в IDE (согласно инструкции выше).

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

4) Разработать программный код приложения

Файл с моделью данных (term.dart)

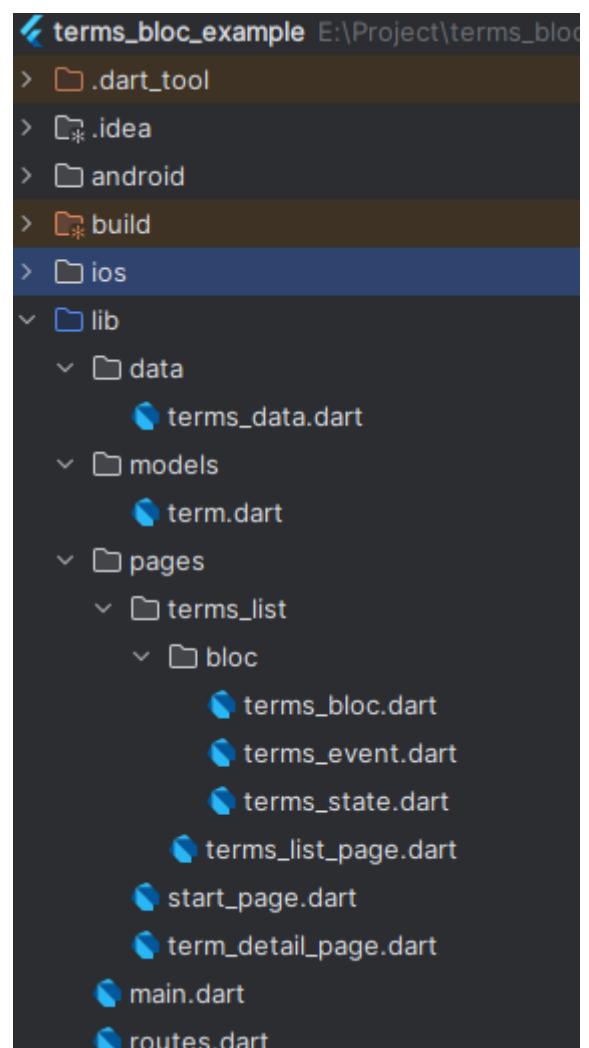
```
class Term {
  final String title;
  final String shortDescription;
  final String fullDescription;

  Term({
    required this.title,
    required this.shortDescription,
    required this.fullDescription,
  });
}
```

Файл с данными в виде коллекции List (terms\_data.dart)

```
import '../models/term.dart';

final List<Term> terms = [
  Term(
    title: 'API',
    shortDescription: 'Интерфейс для взаимодействия программ',
  ),
  Term(
    title: 'HTTP',
    shortDescription: 'Протокол передачи гипертекста',
  ),
  Term(
    title: 'JSON',
    shortDescription: 'Текстовый формат обмена данными',
  ),
]
```



```

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

```

## Файлы BLoC для страницы со списком терминов

Файл с базовым классом событий (terms\_event.dart)

```

part of 'terms_bloc.dart';

@immutable
sealed class TermsEvent {}
// Событие: Загрузить список терминов
final class LoadTermsEvent extends TermsEvent {}

```

Используется sealed class, потому что это позволяет компилятору проверить, что мы обработали все возможные состояния в switch-выражениях. Это предотвращает ошибки и делает код более надёжным.

Файл с базовым классом состояний (terms\_state.dart)

```

part of 'terms_bloc.dart';

@immutable
sealed class TermsState {}

// Ничего не загружено (стартовый экран)
final class TermsInitial extends TermsState {}

// Идет загрузка
final class TermsLoading extends TermsState {}

// Успешная загрузка: получен список терминов
final class TermsLoaded extends TermsState {
    final List<Term> terms;
    TermsLoaded(this.terms);
}

// Ошибка при загрузке
final class TermsError extends TermsState {
    final String message;
    TermsError(this.message);
}

```

Файл с логикой блока (terms\_bloc.dart)

```

import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:terms_bloc_example/data/terms_data.dart';
import 'package:terms_bloc_example/models/term.dart';
import 'package:meta/meta.dart';

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

// Наследование от класса Bloc
class TermsBloc extends Bloc<TermsEvent, TermsState> {

```

```

// Начальное состояние при создании блока - ничего не загружено
TermsBloc(): super(TermsInitial()) {
    // Регистрируется обработчик для события LoadTermsEvent
    on<LoadTermsEvent>(_onLoadTerms);
}
//
Future<void> _onLoadTerms(
    LoadTermsEvent event,
    Emitter emit,
) async {
    // меняем состояние на индикатор загрузки
    emit(TermsLoading());
    await Future.delayed(const Duration(milliseconds: 500)); // имитация загрузки
    //возвращаем новое состояние со списком терминов
    emit(TermsLoaded(terms));
}
}

```

## Файлы слоя UI

Файл со страницей списка терминов (terms\_list\_page.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';

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

    @override
    Widget build(BuildContext context) {
        return Scaffold(
            appBar: AppBar(title: const Text('Список ИТ-терминов')),
            body: BlocBuilder<TermsBloc, TermsState>( //Перестройка UI при изменении состояния
                builder: (context, state) {
                    if (state is TermsLoading) { //состояние - идет загрузка
                        return const Center(child: CircularProgressIndicator());
                    } else if (state is TermsError) { // состояние - ошибка
                        return Center(child: Text(state.message));
                    } else if (state is TermsLoaded) { // состояние - список загружен
                        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,
                                                '/detail',
                                                arguments: term,
                                            );
                                        },
                                    ),
                                );
                            },
                        );
                    }
                }
            );
        }
    }
}

```

```

        // начальное состояние / fallback
        return const Center(child: Text('Загрузка данных...'));
    },
),
);
}
}

```

Файл со начальной страницей – кнопка Показать (start\_page.dart)

```

import 'package:flutter/material.dart';
import 'package:terms_bloc_example/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 детализации термина (term\_detail\_page.dart)

```

import 'package:flutter/material.dart';
import '../models/term.dart';

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),
                    ),
            ),
        );
    }
}

```

Файл с классом маршрутов (routes.dart)

```

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

```

### Файл 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';

void main() {
  runApp(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(),
        // Создается новый блок и событие добавляется в поток
        AppRoutes.list: (context) => BlocProvider(
          create: (_) => TermsBloc()..add(LoadTermsEvent()),
          child: const TermsListPage(),
        ),
        AppRoutes.detail: (context) => const TermDetailPage(),
      },
    );
  }
}
```

При каждом заходе на страницу /list создаётся новый Bloc (bloc живёт только пока открыт этот экран).

Каскадный оператор Dart (...) позволяет сразу вызвать метод на только что созданном объекте, т.е. TermsBloc()..add(LoadTermsEvent()) означает, что создается блок TermsBloc() и сразу у него вызывается метод add(LoadTermsEvent()).

Если список терминов нужен в нескольких местах (не только на этом экране), можно перенести BlocProvider выше, например, в void main():

```
void main() {
  runApp(
    BlocProvider(
      create: (_) => TermsBloc()..add(LoadTermsEvent()),
      child: const MyApp(),
    ),
  );
}
```

### 3. Задание на практическую работу

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

- 1) Разработать структуру проекта в соответствии с архитектурой приложения с использованием паттерна BLoC (для страницы HomePage) без подключения внешнего слоя данных.
- 2) Создать класс для описания данных (модель данных).
- 3) Создать файл с данными в виде коллекции List.
- 4) Создать файлы с реализацией BLoC (event, state, bloc) с состояниями: initial, loading, loaded, error.
- 5) Реализовать управление состоянием страницы Home\_page в программном коде с использованием паттерна BLoC.
- 6) В отчете представить цель работы, описание реализованного BLoC с пояснением его работы, скриншот структуры проекта, скриншоты экранов HomePage и DetailPage, программный код приложения.