



МИНОБРНАУКИ РОССИИ

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

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных
технологий

Отчет по практической работе №8

по дисциплине «Разработка кроссплатформенных мобильных приложений»

Выполнил:

Студент группы ИКБО-07-22

Ларькова Д.Д.

Проверил:

Старший преподаватель кафедры
МОСИТ

Шешуков Л.С.

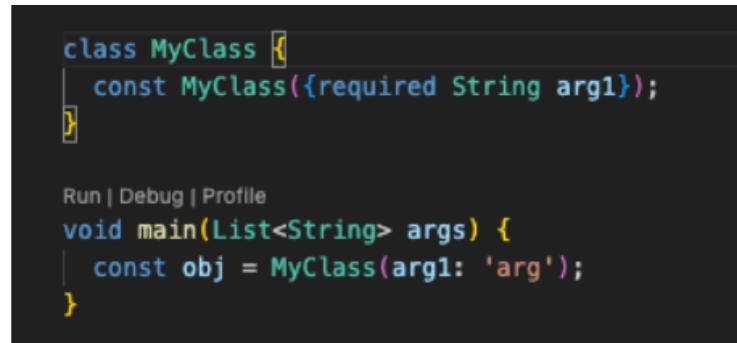
Москва 2025 г.

Цели практической работы:

- Изучить проблему зависимостей в проектах
- Произвести работу с Inherited Widget
- Произвести подключение и работу с DI контейнером GetIt
- Выполнение практической работы №8

1. Проблема зависимостей в проекте

Ранее в практических работах мы уже сталкивались с необходимостью передавать один или более параметр в конструктор при создании объекта того или иного класса. Бывают ситуации, когда аргументов не много или же все их значения заранее известны, либо вычисляются в данном месте (Рисунок 1). В таких ситуациях создать объект такого класса не составляет никакого труда.



```
class MyClass {  
  const MyClass({required String arg1});  
}  
  
Run | Debug | Profile  
void main(List<String> args) {  
  const obj = MyClass(arg1: 'arg');  
}
```

Рисунок 1 – Класс с «простым» конструктором

Однако бывают ситуации, когда перечень аргументов становится многочисленным, а также значения аргументов конструктора уже невозможно рассчитать на месте и их приходится брать из других мест (Рисунок 2). В таких ситуациях приходится делать место, в котором будет создаваться объект такого сложного класса зависимым от тех данных, которые необходимы в конструкторе. При достижении определенного уровня такой вложенности становится все тяжелее взаимодействовать с программным кодом и требуется решать такого рода проблемы.

```
class MyClass {  
    const MyClass({  
        required String arg1,  
        required String arg2,  
        required String arg3,  
        required String arg4,  
        required String arg5,  
        required String arg6,  
        required String arg7,  
        required String arg8,  
        required String arg9,  
        required String arg10,  
    });  
}  
  
Run | Debug | Profile  
void main(List<String> args) {  
    const obj = MyClass(  
        arg1: 'arg',  
        arg2: 'arg2',  
        arg3: 'arg3',  
        arg4: 'arg4',  
        arg5: 'arg5',  
        arg6: 'arg6',  
        arg7: 'arg7',  
        arg8: 'arg8',  
        arg9: 'arg9',  
        arg10: 'arg10',  
    );  
}
```

Рисунок 2 – Конструктор с большим перечнем полей

Самым простым способом решения проблемы поочередной зависимости классов от какого-либо набора данных является вынос этих данных во внешнее пространство, к которому у этих классов будет доступ. Это позволяет не передавать зависимость из класса в класс, а получать ее напрямую из хранилища. Такой принцип называется – внедрение зависимостей или, как в дальнейшем будет называться, DI. В фреймворке Flutter есть разные способы реализации DI, однако рассмотрено будет 2 из них: Inherited Widget и DI контейнер GetIt.

2. Inherited Widget

Inherited Widget является одним из основных типов Widget-ов в фреймворке Flutter. Он в основном выступает в качестве внешнего хранилища данных, располагаемых в дереве виджетов, как любой другой Widget. В отличии от Stateless и Stateful Widget, Inherited Widget не имеет и не предполагает какого-либо отображения на экране, поэтому имеет отличные как внутреннее устройство, так и принцип взаимодействия.

Все Widget-ы, зависящие от данных в Inherited Widget подписываются

на его состояние и получают данные по структуре Widget-ов. Если Widget имеет собственный State, то его жизненный цикл подразумевает автоматическое обновление при обновлении данных в Inherited Widget при помощи метода didChangedDependencies. Однако получить данные без автоматической актуализации данных может любой виджет, у которого есть собственный context. По нему Widget ищет объект класса наследника Inherited Widget выше по дереву и возвращает объект этого класса, если такой был найден.

2.1. Статический метод of

Основным требованием к хранилищу зависимостей является доступ к нему из разных мест приложения. Данное требование в Inherited Widget и его наследниках реализуется посредством реализации статического метода of. Данный метод позволяет найти объект класса Inherited Widget или его наследника в виджете дерева выше Widget-а, в котором необходима зависимость. В связи с тем, что объект класса Inherited Widget или его наследника будет искааться в структуре Widget-ов, то для его корректной логики требуется context Widget-а, в котором необходима зависимость. Пример синтаксиса статического метода of проиллюстрирован на рисунке 3.

```
static MyLogic of(BuildContext context)
```

Рисунок 3 – сигнатура статического метода of

2.2. Метод dependOnInheritedWidgetOfExactType

Для поиска объекта класса Inherited Widget или его наследников фреймворк предоставляет специальный метод dependOnInheritedWidgetOfExactType, который возвращает объект класса, указанный в качестве дженерик типа в случае, если объект был найден выше по дереву, или пустоту, если объекта данного класса обнаружено не было. Важно заметить, что этот метод используется для поиска объекта по дереву, а значит его нужно вызывать у context, объекта класса BuildContext. Именно для этого метод of обязательно должен иметь в качестве аргумента context Widget-

а, в котором нужна зависимость. Полную реализацию метода `of` можно увидеть на рисунке 4.

Важно заметить, что метод `dependOnInheritedWidgetOfExactType` вернет первый найденный объект класса, указанный в качестве дженерик класса. То есть если в дереве Widget-ов находятся два Widget-а одного Inherited Widget-а или его наследника, то метод `dependOnInheritedWidgetOfExactType` вернет именно ближайший Widget, расположенный верх по дереву от зависимого от данных Widget-а.

```
static MyLogic of(BuildContext context) =>
    context.dependOnInheritedWidgetOfExactType<MyLogic>()!;
```

Рисунок 4 – использование метода `dependOnInheritedWidgetOfExactType` в статическом методе `of`

2.3. Метод `updateShouldNotify`

Так как Inherited Widget и его наследники используются для передачи набора данных в Widget-ы, где эти данные требуются, то они также должны иметь реализацию автоматического оповещения этих виджетов при обновлении хранимых данных. Этот механизм реализован при помощи метода `updateShouldNotify`, возвращающего логическое значение, которое описывает необходимость слушателям этих данных обновить свои состояния, так как данные обновились. Разработчик вправе сам регулировать, когда и при каких условиях слушатели обновят свои состояния, именно регулируя логику возвращения данного логического значения, сравнивая экземпляр класса до изменения и после изменения. Фреймворк самостоятельно сохраняет состояние объекта до изменения и предоставляет его в данный метод через аргумент `oldWidget`. Пример реализации логики метода `updateShouldNotify` можно увидеть на рисунке 5.

```
@override
bool updateShouldNotify(covariant MyLogic oldWidget) =>
    oldWidget.number != number;
```

Рисунок 5 – реализация метода `updateShouldNotify` на основе внутреннего поля `number`

3. DI контейнер GetIt

GetIt – это DI контейнер, который распространяется в одноименном пакете. Это простой DI контейнер для проектов Dart и Flutter с некоторыми дополнительными преимуществами. Его можно использовать вместо InheritedWidget для доступа к объектам, например, из вашего пользовательского интерфейса. Главное отличие GetIt от Inherited Widget заключается в том, что для доступа к данным не требуется context Widget-а, от которого происходит обращение. Очень часто в ходе разработки требуется получить данные в месте, где нет доступа к context и в таких моментах Inherited Widget становится очень неудобен. GetIt отлично закрывает эту проблему, так как он не является частью фреймворка Flutter и не зависит от его внутренней реализации. GetIt строится на объектном паттерне Singleton, который позволяет получить доступ к единому экземпляру классу в любой точке приложения без необходимости как-то добавлять его в дерево Widget-ов.

3.1. Подключение в проект

Для подключения пакета get_it в проект требуется в файл pubspec.yaml добавить пакет в dependency. Пример добавления изображен на рисунке 6.

```
dependencies:  
  get_it: ^7.7.0
```

Рисунок 6 – добавления пакета get_it в приложение

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

3.2. Доступ к DI контейнеру

Для того, чтобы получить доступ к DI контейнеру GetIt, необходимо обратиться к его статичному полю instance или полю I. Данное поле хранит в себе единый объект DI контейнера, используемого в приложении. Получив доступ к объекту GetIt, разработчику открывается полный доступ к контейнеру и лежащих в нем образах. Способ обращения к контейнеру можно

рассмотреть на рисунке 7.

```
GetIt getIt = GetIt.instance;  
//There is also a shortcut (if you don't like it just ignore it):  
GetIt getIt = GetIt.I;
```

Рисунок 7 – пример обращения к контейнеру GetIt

3.3. Регистрация объектов

Для того, чтобы некий образ класса появился в контейнере его необходимо сначала там зарегистрировать. В контейнере можно зарегистрировать образ как уже готовый объект или как алгоритм его создания. Для регистрации объектов в контейнере GetIt есть специализированный метод registerSingleton<T>. Указывая в качестве значения его аргумента объект дженерик класса, он регистрируется в контейнере и будет возвращаться каждый раз, когда из контейнера будут запрашивать образ данного класса. (Рисунок 8)

```
// if you want to work just with the singleton:  
GetIt.instance.registerSingleton<AppModel>(AppModelImplementation());  
GetIt.I.registerLazySingleton<RESTAPI>(() => RestAPIImplementation());
```

Рисунок 8 – регистрация объектов в контейнер GetIt

Иногда требуется хранить более одного объекта одного и того же класса в контейнере, тогда при регистрации объекта помимо самого объекта передают его идентификационное название, по которому в дальнейшем можно будет получить именно этот образ запрашиваемого класса. (Рисунок 9)



```
Run | Debug | Profile  
void main(List<String> args) {  
    GetIt.I.registerSingleton(MyClass(), instanceName: 'my_class');  
}
```

Рисунок 9 – указание идентификационное имени объекта в контейнере GetIt

При стандартной настройке контейнера если попытаться зарегистрировать объект уже имеющегося в контейнере класса, то будет получена runtime ошибка. Для решения этой проблемы можно либо задать каждому экземпляру собственное идентификационное название или включить настройку переписи объектов внутри контейнера. (Рисунок 10) Она позволяет заменять в контейнере объект одного класса без ручной очистки образа класса.

```
Run | Debug | Profile
void main(List<String> args) {
    GetIt.I.allowReassignment = true;
}
```

Рисунок 10 – включение настройки переписи объектов в контейнере GetIt

Так как регистрация объектов в контейнере – это очень затратный процесс по памяти, GetIt имеет возможность произвести «отложенную» регистрацию. Для этого необходимо вызвать метод registerLazySingleton<T>, в аргументы которого передается функция, возвращающая требуемых объект дженерик класса. (Рисунок 8) Это позволяет заложить в контейнер не сам объект, а функцию его получения, что уменьшает потребляемую память и позволяет заложить в контейнер объект на момент первого обращения к образу класса.

3.4. Регистрация фабрик создания образов

Не во всех случаях требуется хранить объект класса в контейнере. Существуют подходы, что классы описывают логику работы алгоритмов, однако не хранят в своих объектах никаких данных. Так как хранение объектов в контейнере очень затратный процесс по памяти, то хранить такого рода классы нет никакой необходимости. Вместо этого, контейнер позволяет сохранять в качестве образа не сам объект, а логику получения образа класса. Такие функции с логикой создания объектов требуемых классов называются фабриками. Для регистрации фабрики в качестве образа класса в контейнере используется метод registerFactory<T>. (Рисунок 11) При обращении в контейнер за образом класса будет каждый раз получено новый объект данного класса, построенный по логике, заложенной в фабрике.

Так же при регистрации фабрики ей можно задать идентификационное название, чтобы отличать различные фабрики, возвращающие образ одного и того же класса.

```
Run | Debug | Profile
void main(List<String> args) {
    GetIt.I.registerFactory(() -> MyClass(), instanceName: 'my_class');
}
```

Рисунок 11 – регистрация фабрики в контейнер GetIt

3.5. Получение образа из контейнера

В ситуации, когда требуется получить образ из контейнера, требуется использовать специализированный метод `get<T>`, который вернет образ запрашиваемого дженерик класса. (Рисунок 12) Если образ не был ранее зарегистрирован в контейнере, то данный метод вернет runtime ошибку, что такого образа в контейнере нет.

```
// as Singleton:  
var myAppModel = GetIt.instance<AppModel>();  
var myAppModel = GetIt.I<AppModel>();
```

Рисунок 12 – получение образа из контейнера GetIt

3.6. Проверка на наличие образа в контейнере

Для того, чтобы быть уверенным в получении образа искомого класса из контейнера, GetIt позволяет проверять, зарегистрирован ли образ класса до того, как его получать. Для проверки регистрации образа используется метод `isRegistered<T>`. (Рисунок 13) Данный метод возвращает логическое значение, обозначающее регистрацию образа дженерик класса в контейнере. Так же, как и в методах регистрации, при помощи дополнительного аргумента можно указать идентификационное название для проверки регистрации конкретного образа дженерик класса.

```
Run | Debug | Profile  
void main(List<String> args) {  
| GetIt.I.isRegistered<MyClass>(instanceName: 'my_class');  
}
```

Рисунок 13 – регистрация фабрики в контейнер GetIt

4. Выполнение практической работы №8

В рамках выполнения практической работы №8 было переработано существующее приложение для учета игр. Основная цель — реализация передачи параметров (в частности, доступа к репозиторию игр) по всему приложению с использованием двух изученных способов: Inherited Widget и DI контейнера GetIt.

Переработка выполнялась в два этапа:

- Реализация с Inherited Widget (контекст-зависимый доступ).
- Реализация с GetIt (глобальный доступ без контекста).

Для каждого способа приведены изменения в коде, примеры использования и объяснения. Приложение остается функциональным: добавление, редактирование, удаление работают с обновлением UI через setState или навигацию.

4.1. Реализация с Inherited Widget

Inherited Widget — это встроенный механизм Flutter для передачи данных (включая зависимости) вниз по дереву виджетов без необходимости ручной передачи через конструкторы. В данном случае Inherited Widget используется для распространения экземпляра NoteRepository по всему приложению. Репозиторий создается на верхнем уровне (в GameTracker) и оборачивается в кастомный Inherited Widget. Доступ к репозиторию осуществляется через статический метод of(context), который требует наличияBuildContext, что делает этот подход тесно интегрированным с Flutter-структурой.

Этот метод идеален для небольших и средних Flutter-приложений, где все логика сосредоточена в виджетах.

Изменения в GameService

Для обеспечения совместимости с паттерном инъекции зависимостей GameService реализован как экземплярный класс без статических методов и полей. Это позволило управлять экземпляром сервиса вручную и передавать его через GameScope, основанный на InheritedWidget.

Внутри класса определены методы addGame, deleteGame, toggleStatus, deleteCompleted, а также экземплярный список games, который содержит все игры приложения. Экранные компоненты больше не обращаются к глобальным данным, а работают с экземпляром сервиса, полученным из контекста. Такой подход соответствует принципу Dependency Inversion, поскольку представление зависит от абстракции сервиса, а не от его конкретной реализации или глобального состояния. (Рисунок 14).

```
class GameService {
    final List<Game> _games = [
        > Game(...),
        > Game(...),
        > Game(...),
        > Game(...),
        > Game(...),
    ];

    List<Game> get games => _games;

    List<Game> getAllGames() => _games;

    void addGame(Game game) => _games.add(game);

    Game? getById(String id) {
        try {
            return _games.firstWhere((g) => g.id == id);
        } catch (_) {
            return null;
        }
    }

    void updateGame(String id, Game updated) {
        final index = _games.indexWhere((g) => g.id == id);
        if (index != -1) _games[index] = updated;
    }

    void deleteGame(String id) {
        _games.removeWhere((g) => g.id == id);
    }
}
```

Рисунок 14 - GameService без static

Создание Inherited Widget

Для распространения зависимости по дереву приложения был создан новый файл game_scope.dart, содержащий класс GameScope, наследующий StatelessWidget, и вложенный InheritedWidget _GameInherited.

GameScope хранит экземпляр GameService и предоставляет статические методы of(context) и расширение context.gameService для доступа к данным из любой части дерева виджетов.

При изменении состояния в GameScopeState вызывается setState(), в результате чего обновляется вложенный InheritedWidget, который оповещает все зависимые виджеты о произошедших изменениях. Это гарантирует автоматическое обновление пользовательского интерфейса при изменении списка игр или изменении их статусов. Таким образом, GameScope выступает в роли простого механизма передачи состояния приложения.

```
import 'package:flutter/widgets.dart';
import 'game_service.dart';

class GameScope extends InheritedWidget {
    final GameService gameService;

    const GameScope({
        super.key,
        required this.gameService,
        required super.child,
    });

    static GameScope of(BuildContext context) {
        final scope =
            context.dependOnInheritedWidgetOfExactType<GameScope>();
        assert(scope != null, 'GameScope не найден в контексте');
        return scope!;
    }

    @override
    bool updateShouldNotify(GameScope oldWidget) =>
        !identical(gameService, oldWidget.gameService);
}

extension GameScopeX on BuildContext {
    GameService get gameService => GameScope.of(this).gameService;
}
```

Рисунок 15 - Класс GameScope

Интеграция в GameTracker

На верхнем уровне приложения, в файле main.dart, создаётся экземпляр GameService, после чего весь MaterialApp.router обрамляется в GameScope, которому этот экземпляр передаётся через параметр gameService (рисунок 16).

Благодаря этому подходу единый экземпляр GameService становится доступен всем виджетам ниже по дереву, а доступ к состоянию можно получить через GameScope.of(context) или удобное расширение context.gameService.

Такое решение гарантирует использование одного общего состояния для всего приложения и при этом остаётся достаточно гибким.

```
void main() {
  runApp(
    const GameScope(
      child: GameTracker(),
    ), GameScope
  );
}

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

  @override
  Widget build(BuildContext context) {
    final router = buildRouter();
    return MaterialApp.router(
      debugShowCheckedModeBanner: false,
      title: 'Games',
      theme: ThemeData(
        useMaterial3: true,
        colorSchemeSeed: Colors.indigo,
      ), ThemeData
      routerConfig: router,
    ); MaterialApp.router
  }
}
```

Рисунок 16 - Оборачивание приложения в GameScope для распространения зависимости

Использование в экранах

В экранах приложения (GameListScreen, AddEditGameScreen, GameDetailScreen, StatsScreen) прямые и статические обращения к GameService были заменены на доступ через GameScope.of(context).repository. Такой подход обеспечивает инъекцию зависимости без необходимости передавать экземпляр сервиса через конструкторы, что делает код чище и соответствует принципам Dependency Injection.

В экране StatsScreen отображение статистики игр также основано на доступе к общему состоянию приложения, размещённому внутри InheritedWidget.

```

Widget build(BuildContext context) {
    final repo = GameScope.of(context).repository;
    final games = repo.games;

    final completed = games.where((g) => g.status == 'Completed').length;
    final playing = games.where((g) => g.status == 'Playing').length;
    final planned = games.where((g) => g.status == 'Planned').length;

    return Scaffold(
        appBar: AppBar(title: const Text('Статистика')),
        body: Padding(
            padding: const EdgeInsets.all(16),
            child: Column(
                crossAxisAlignment: CrossAxisAlignment.start,
                children: [
                    Text('Всего игр: ${games.length}', style: const TextStyle(fontSize: 18)),
                    const SizedBox(height: 8),
                    Text('Завершено: $completed'),
                    Text('Игрюю: $playing'),
                    Text('Запланировано: $planned'),
                ],
            ),
        ),
    );
}

```

Рисунок 17 — Фрагмент метода `build` в модифицированный под использование `GameScope`.

Далее можем убедиться в корректности работы данной модификации посмотрев на состояния экранов на рисунках 18-19.

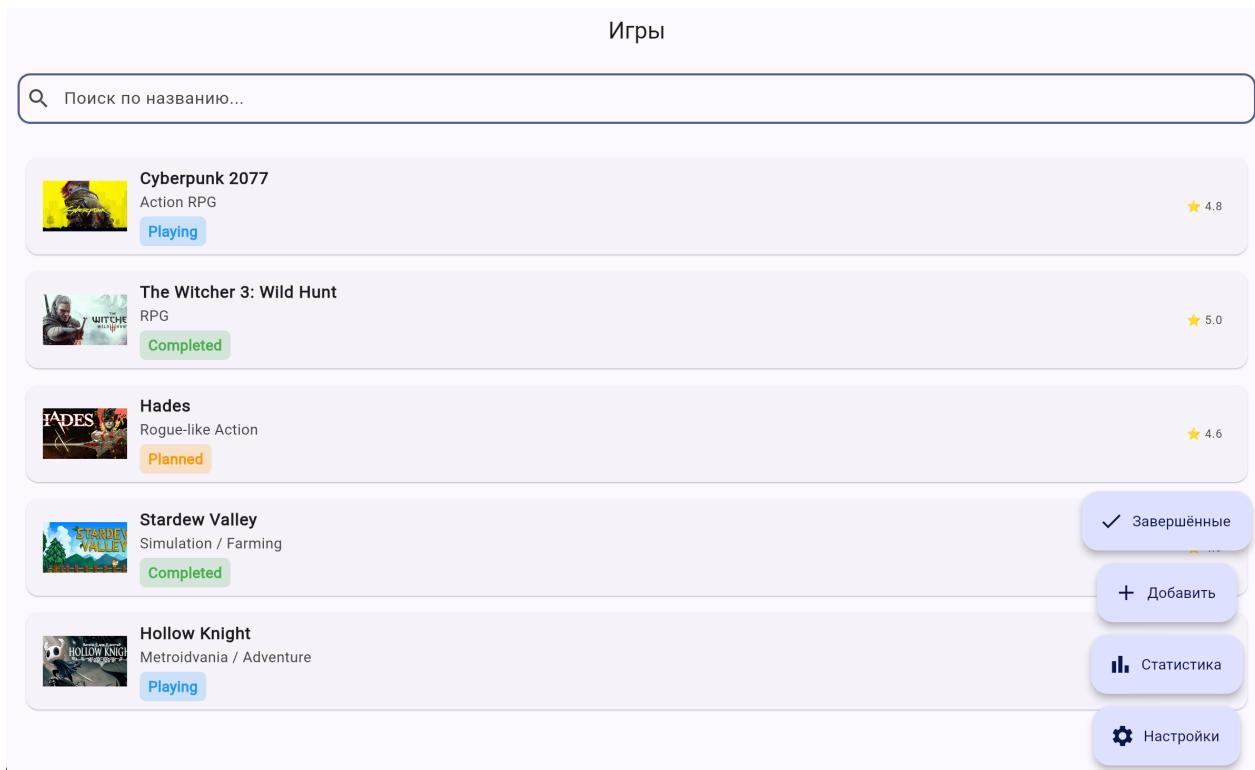


Рисунок 18 — Экран списка игр с различными статусами

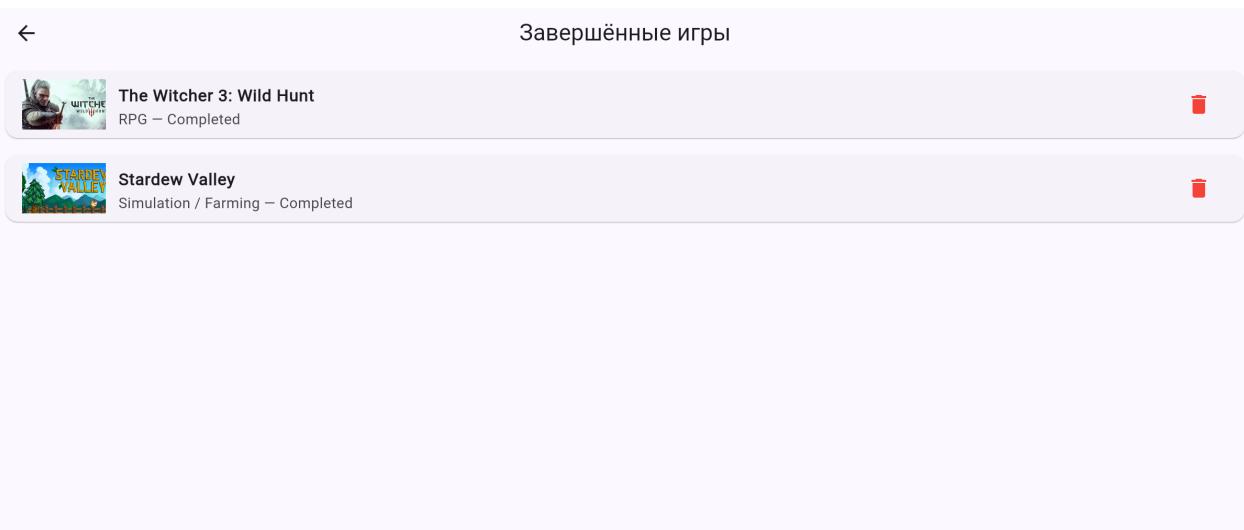


Рисунок 19 — Экран списка законченных игр после перехода на него

В экране CompletedGamesScreen удаление отдельной карточки завершённой игры реализовано через доступ к общему состоянию приложения с использованием механизма InheritedWidget. Непосредственно в обработчике удаления элемента списка вызывается: GameScope.read(context).repository.deleteGame(game.id);GameScope.read(context).state.notify(); где GameScope — это созданная в приложении Inherited-обёртка, предоставляющая доступ к единому экземпляру GameService, содержащему общий список всех игр.

Обработчик получает идентификатор удаляемой игры и передаёт его в метод deleteGame(). После удаления изменение фиксируется в общем состоянии через внутренний setState, вызываемый внутри GameScope методом notify().

Благодаря этому зависимые виджеты автоматически уведомляются об изменении состояния. После перестройки виджета метод build() экрана повторно выполняет фильтрацию и список отображается уже без удалённой игры.

Таким образом, удаление завершённой игры производится централизованно — непосредственно из общего списка, без использования локальных копий данных и без проброса сервиса через конструкторы. Актуальное состояние мгновенно отражается в интерфейсе благодаря механизму обновления, основанному на InheritedWidget, что подтверждает

корректную реализацию реактивной модификации данных.

```
return Card(
    margin: const EdgeInsets.all(8),
    child: ListTile(
        leading: Image.network(
            game.imageUrl,
            width: 80,
            height: 80,
            fit: BoxFit.cover,
        ), Image.network
        title: Text(
            game.title,
            style: const TextStyle(fontWeight: FontWeight.bold),
        ), Text
        subtitle: Text('${game.genre} - ${game.status}'),
        trailing: IconButton(
            icon: const Icon(Icons.delete, color: Colors.red),
            onPressed: () {
                GameScope.read(context).repository.deleteGame(game.id);
            }
        )
    )
)
```

Рисунок 20 — Обработчик удаления карточки

Далее можем убедиться в корректности работы данной модификации посмотрев на состояния экранов на рисунках 21–22.

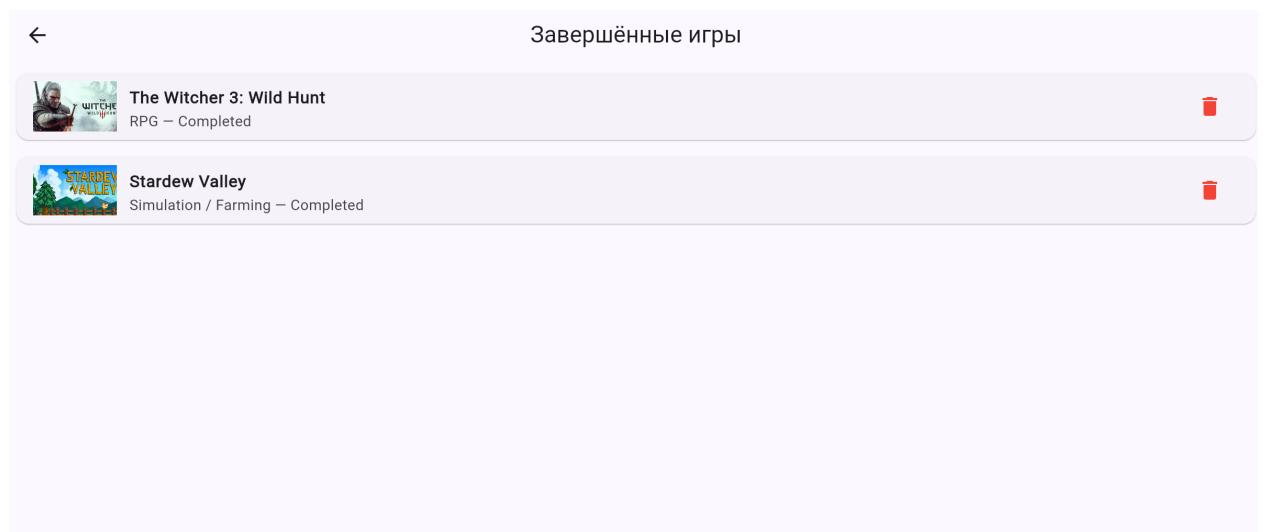


Рисунок 21 — Экран списка законченных игр до удаления выбранной карточки.

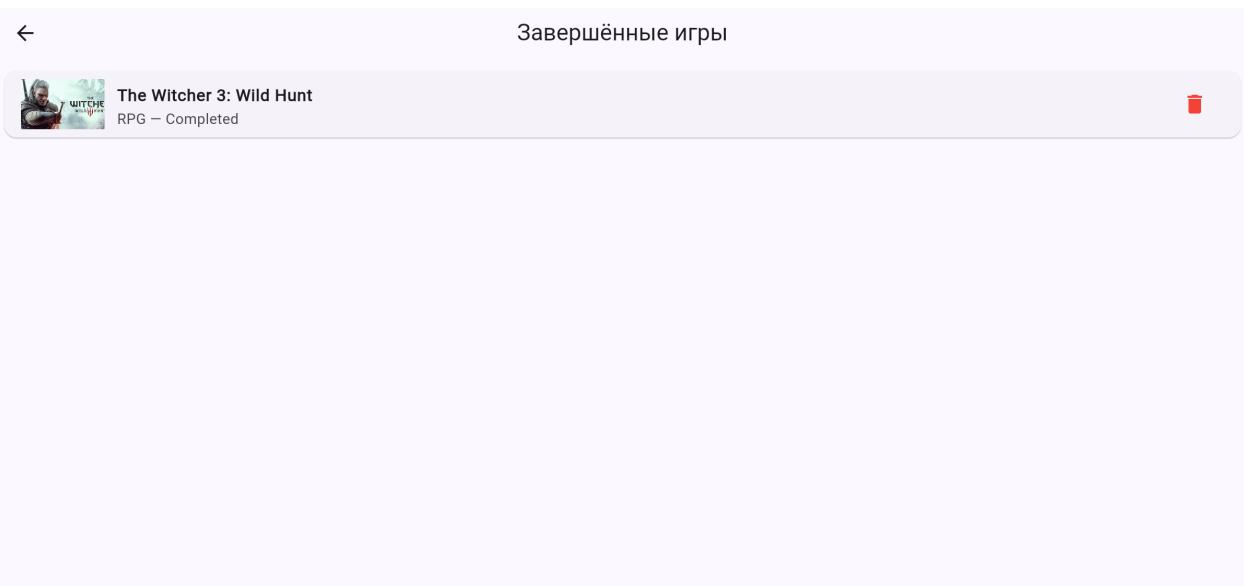


Рисунок 22 — Экран законченных игр после удаления.

В экране GameListScreen отрисовка списка игр реализована через доступ к общему состоянию приложения с использованием механизма InheritedWidget.

После получения коллекции применяется локальная фильтрация по поисковой строке, привязанной к TextEditingController. Введённая пользователем строка приводится к нижнему регистру и сравнивается с названием игры. В результате в интерфейс передаётся только отфильтрованная коллекция, что позволяет динамически обновлять список игр при каждом изменении поля поиска. Рисунок 23 демонстрирует фрагмент метода build, в котором выполняется фильтрация списка игр..

```
final TextEditingController _searchController = TextEditingController();
```

```
List<Game> _filteredGames() {
    final query = _searchController.text.toLowerCase();
    final repo = GameScope.of(context).repository;

    return repo.games
        .where((g) => g.title.toLowerCase().contains(query))
        .toList();
}

@Override
Widget build(BuildContext context) {
    final games = _filteredGames();
```

Рисунок 23 — Фрагмент метода build в GameListScreen.

Далее можем убедиться в корректности работы данной модификации

посмотрев на состояния экранов на рисунках 24–25.

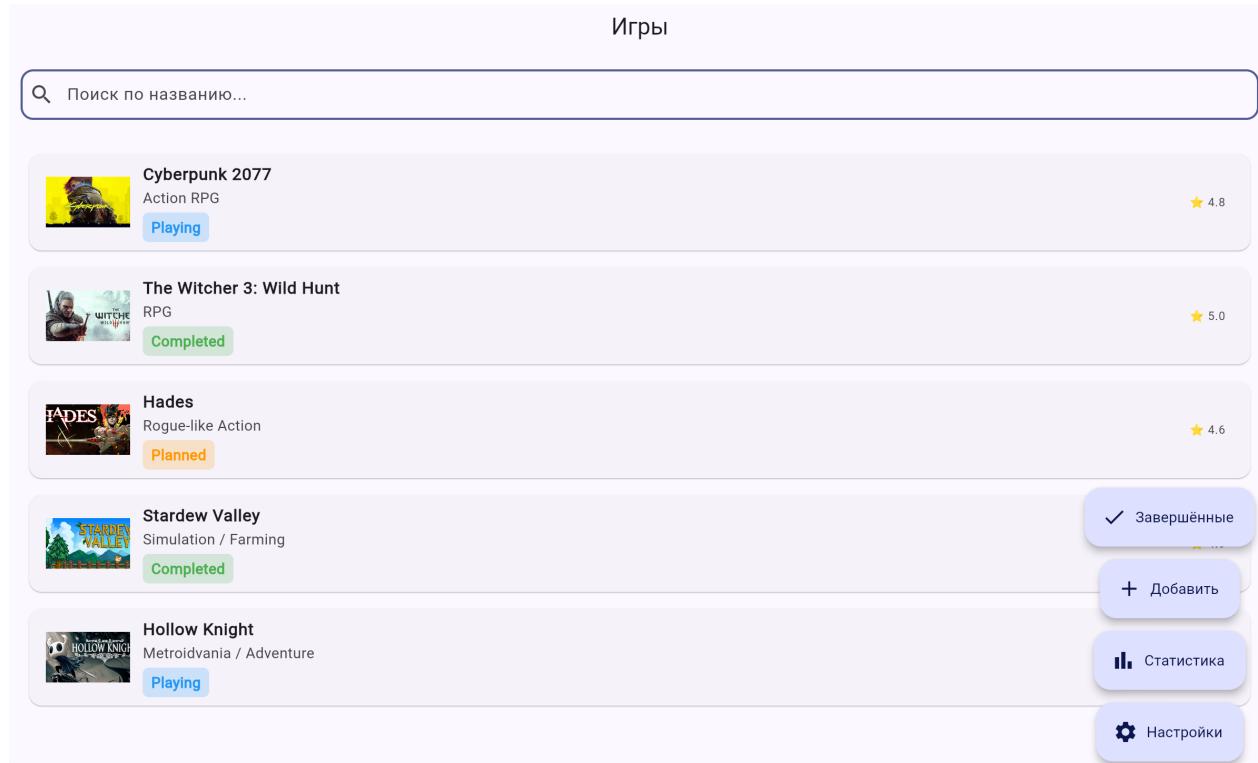


Рисунок 24 — Экран списка игр без фильтрации.

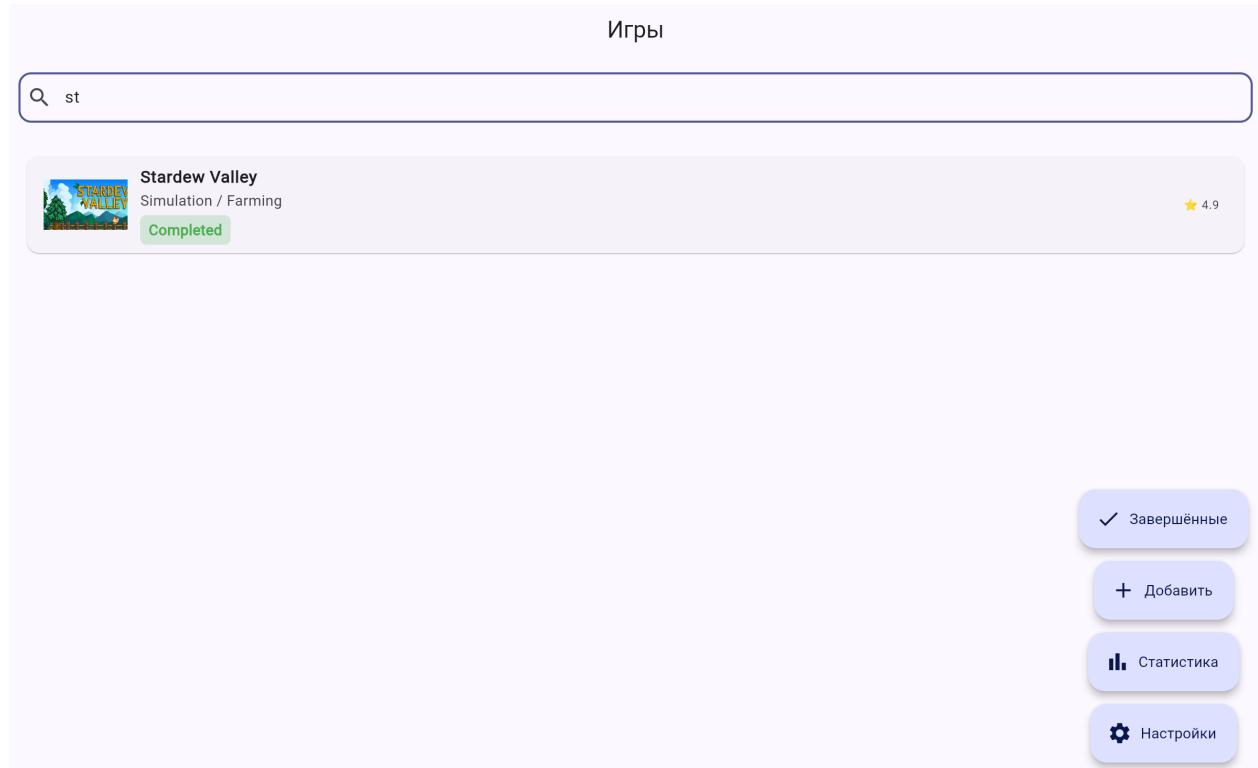


Рисунок 25 — Экран списка игр после применения поиска.

В экране AddEditGameScreen добавление новой игры реализовано через доступ к общему состоянию приложения с использованием InheritedWidget. Логика создания игры находится в обработчике нажатия кнопки «Сохранить»:

после заполнения пользователем всех необходимых полей (название, жанр и т. д.) формируется новый экземпляр модели Game, в который передаются введённые значения (рисунок 26).

Затем, через обращение к глобальному состоянию, полученному из GameScope, созданная игра добавляется в общий список.

```
void _save() {
    final repo = GameScope.read(context).repository;

    final game = Game(
        id: DateTime.now().millisecondsSinceEpoch.toString(),
        title: _titleController.text.trim(),
        genre: _genreController.text.trim(),
        status: _selectedStatus,
        imageUrl: _imageController.text.trim(),
        rating: double.tryParse(_ratingController.text) ?? 0,
        comment: _commentController.text.trim(),
    ); Game

    repo.addGame(game);
    GameScope.read(context).state.notify();

    context.go("/");
}
```

Рисунок 26 — Фрагмент метода `_save` в `AddEditGameScreen`

Далее можем убедиться в корректности работы данной модификации, посмотрев на состояния экранов на рисунках 27–29.

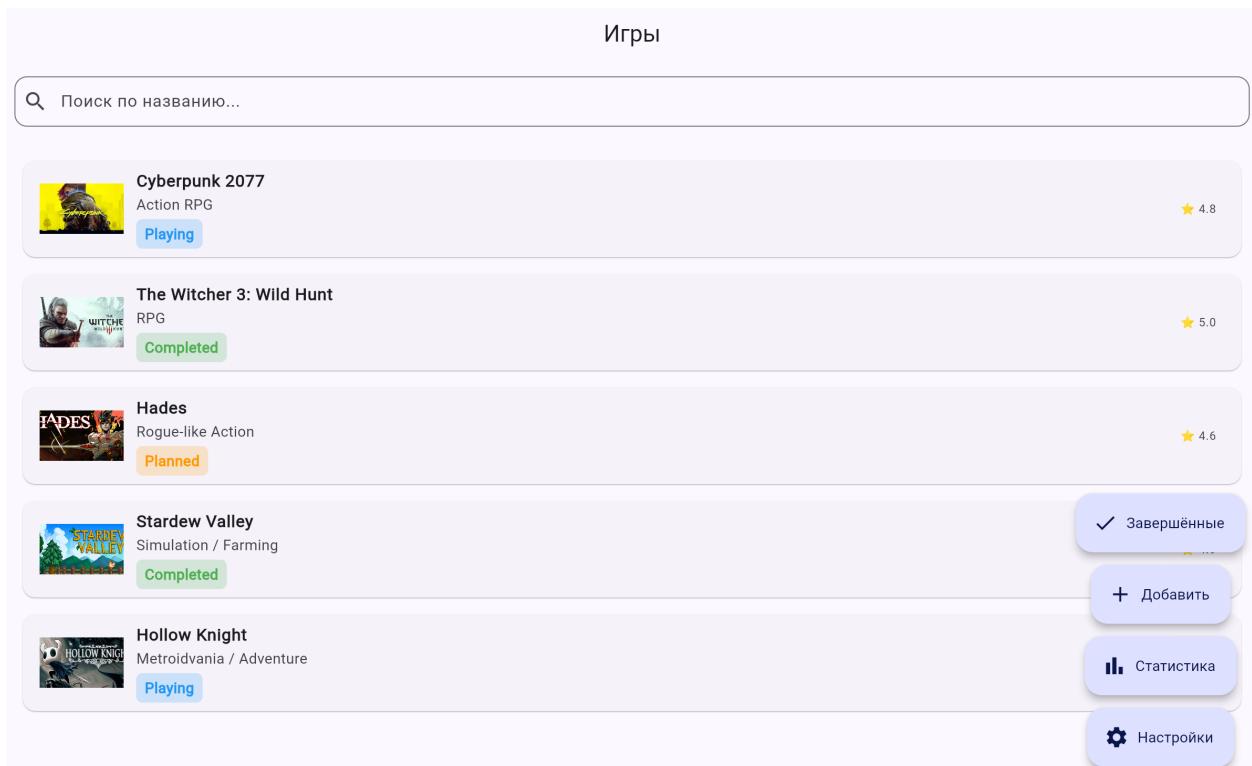


Рисунок 27 — Экран списка игр до добавления новой игры.

The screenshot shows a form for adding a new game:

Название: Factorio

Жанр: Strategy

URL картинки: https://steamcdn-a.akamaihd.net/steam/apps/427520/ss_4d2dbc665c41127ff1e08e4751dce3e396c31318.jpg?t=1530541487

Рейтинг: 7

Комментарий: сложно

Запланировано

Рисунок 28 — Экран добавления игры с заполненными полями.

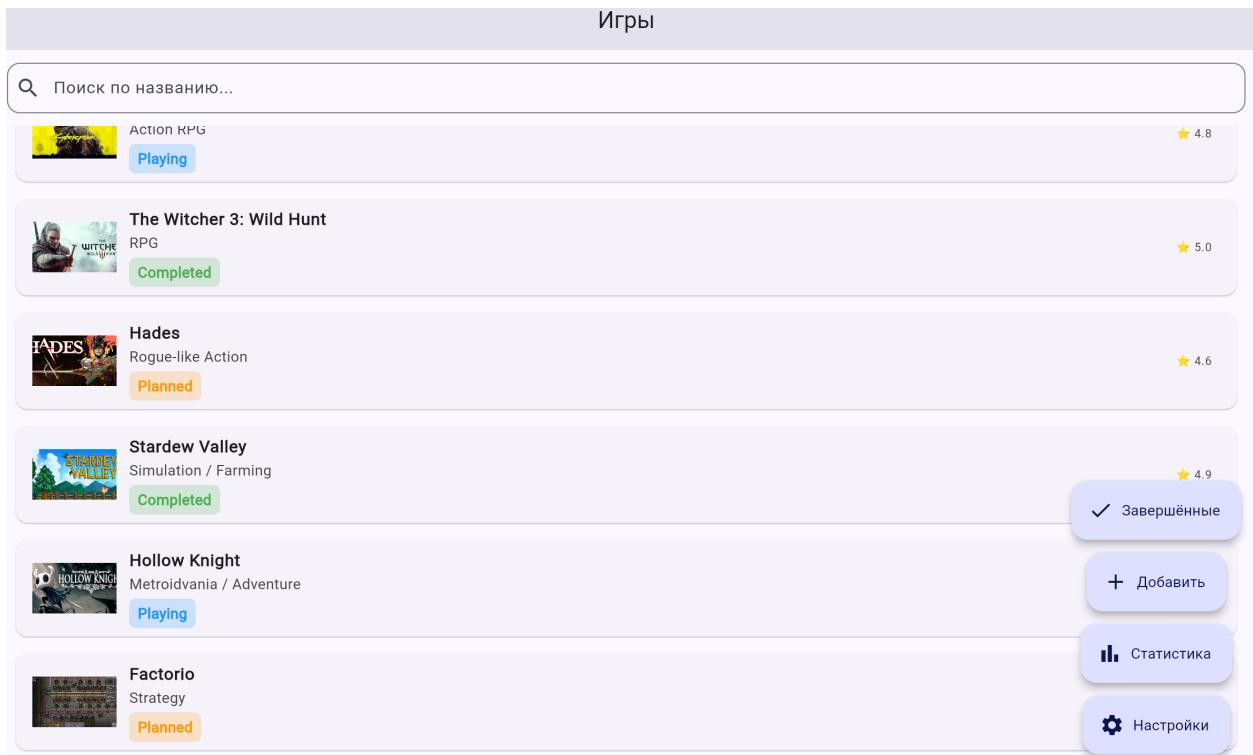


Рисунок 29 — Экран списка игр после добавления новой игры.

4.2. Реализация с DI контейнером GetIt

GetIt — это лёгковесный контейнер внедрения зависимостей (DI-контейнер) из пакета `get_it`, реализующий паттерн Service Locator. Он позволяет регистрировать зависимости глобально и получать к ним доступ из любой части приложения без необходимости передачи `BuildContext`. В данном приложении сервис `GameService`, отвечающий за хранение и управление списком целей, регистрируется как `singleton` в файле `main.dart` перед запуском приложения. Это обеспечивает единый экземпляр сервиса во всём приложении, к которому обращаются все экраны, использующие список целей и операции над ним.

Преимущества использования GetIt: доступ к сервисам без контекста (что удобно для бизнес-логики и сервисов вне виджетов), простота интеграции, поддержка `lazy loading` и фабрик. Недостатки: требуется внешний пакет, нет автоматической подписки на изменения (в отличие от `InheritedWidget`), возможны сложности при тестировании из-за глобального состояния. Данный способ идеально подходит для проектов со структурой, разделённой на слои — UI, бизнес-логика и данные.

Добавление пакеты

Для использования GetIt пакет добавляется в раздел dependencies файла pubspec.yaml (Рисунок 30). Это позволяет импортировать GetIt в коде и использовать его API для регистрации и получения зависимостей. После добавления выполняется команда flutter pub get для установки пакета.

Версия ^7.7.0 выбрана как стабильная на момент разработки.

```
dependencies:  
    cached_network_image: ^3.4.1  
    go_router: ^14.2.7  
    get_it: ^7.7.0  
    flutter:  
        sdk: flutter
```

Рисунок 30 – Добавление get_it в pubspec.yaml

Регистрация в main.dart

Перед вызовом runApp в файле main.dart подключается пакет get_it, после чего создаётся экземпляр сервис-локатора и регистрируется сервис GameService как singleton с помощью метода registerSingleton. Регистрация выполняется при запуске приложения, что гарантирует инициализацию сервиса до построения пользовательского интерфейса. Опционально можно использовать метод registerLazySingleton, если требуется отложенная инициализация, однако в данном случае сервис лёгкий и создаётся сразу.

```
void main() {  
    GetIt.I.registerSingleton<GameService>(GameService());  
  
    runApp(const GameTracker());  
}
```

Рисунок 31 – Регистрация репозитория в GetIt как singleton в main.dart

Использование в экранах

Во всех экранах вызовы GameScope.of(context) были заменены на обращение к сервису напрямую через GetIt.I<GameService>().

Это позволило получать репозиторий игр без необходимости доступа к context, что упрощает код и делает возможным использование сервиса в не-виджет методах.

В экране CompletedGamesScreen отрисовка элементов списка завершённых игр выполнена через доступ к общему состоянию приложения с использованием DI-контейнера GetIt.

Непосредственно в методе build данные извлекаются из зарегистрированного сервиса GameService через вызов: final games = GetIt.I<GameService>().games (рисунок 32).

Далее формируется коллекция завершённых игр с помощью фильтрации. Полученный список передаётся в построение пользовательского интерфейса, где создаётся список карточек завершённых игр, каждая из которых получает экземпляр модели Game и отображает соответствующую информацию — название, жанр, изображение и текущий статус.

```
class _CompletedGamesScreenState extends State<CompletedGamesScreen> {
  @override
  Widget build(BuildContext context) {
    final repo = GetIt.I<GameService>();

    final completed = repo.games
      .where((g) => g.status == 'Completed')
      .toList();
  }
}
```

Рисунок 32 — Фрагмент метода build в CompletedGamesScreen, использующий получение данных через GetIt.I< GameService >()

Далее можем убедиться в корректности работы данной модификации посмотрев на состояния экранов на рисунках 33 - 34.

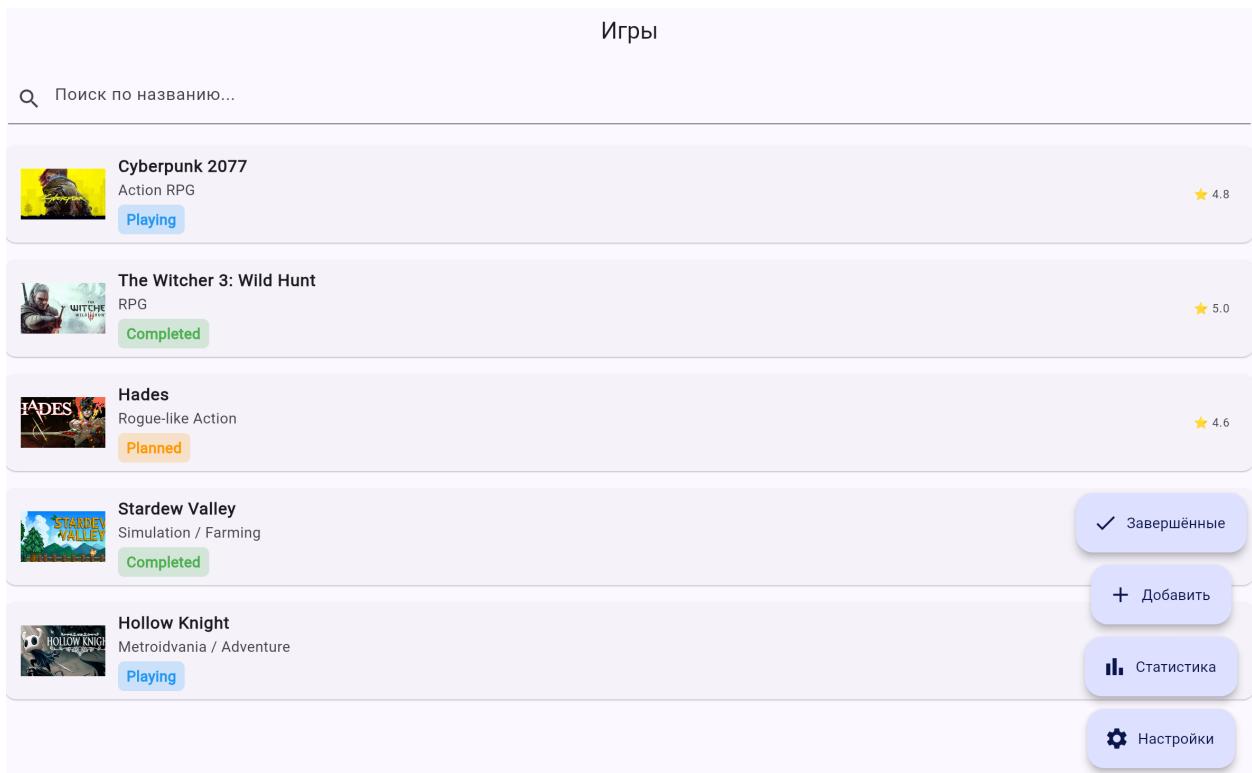


Рисунок 33 — Экран списка игр с различными статусами

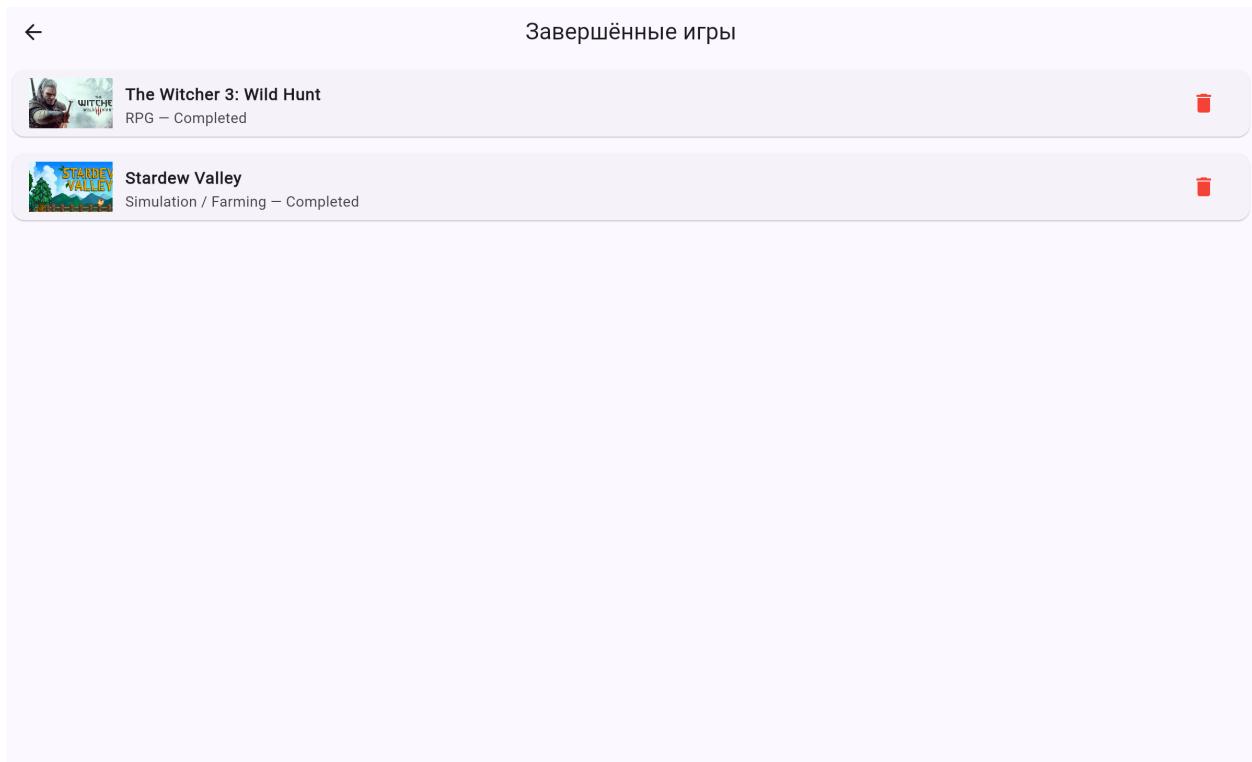


Рисунок 34 — Экран списка законченных игр после перехода на него

В экране CompletedGamesScreen удаление отдельной карточки завершённой игры реализовано через обращение к общему состоянию приложения, предоставляемому DI-контейнером GetIt.

В обработчике удаления элемента списка вызывается метод: `GetIt.I<GameService>().deleteGame(game.id);`, где `GameService` — зарегистрированный в контейнере сервис, отвечающий за хранение и управление списком игр (рисунок 35).

Обработчик получает идентификатор удаляемой игры и передаёт его в метод `deleteGame`, который удаляет соответствующий объект из коллекции `games`.

После выполнения удаления вызывается `setState`, что инициирует повторный вызов метода `build` где производится фильтрация завершённых игр

```
final completed = repo.games
    .where((g) => g.status == 'Completed')
    .toList();
```

Рисунок 35 — Обработчик удаления карточки

Далее можем убедиться в корректности работы данной модификации посмотрев на состояния экранов на рисунках 36–37

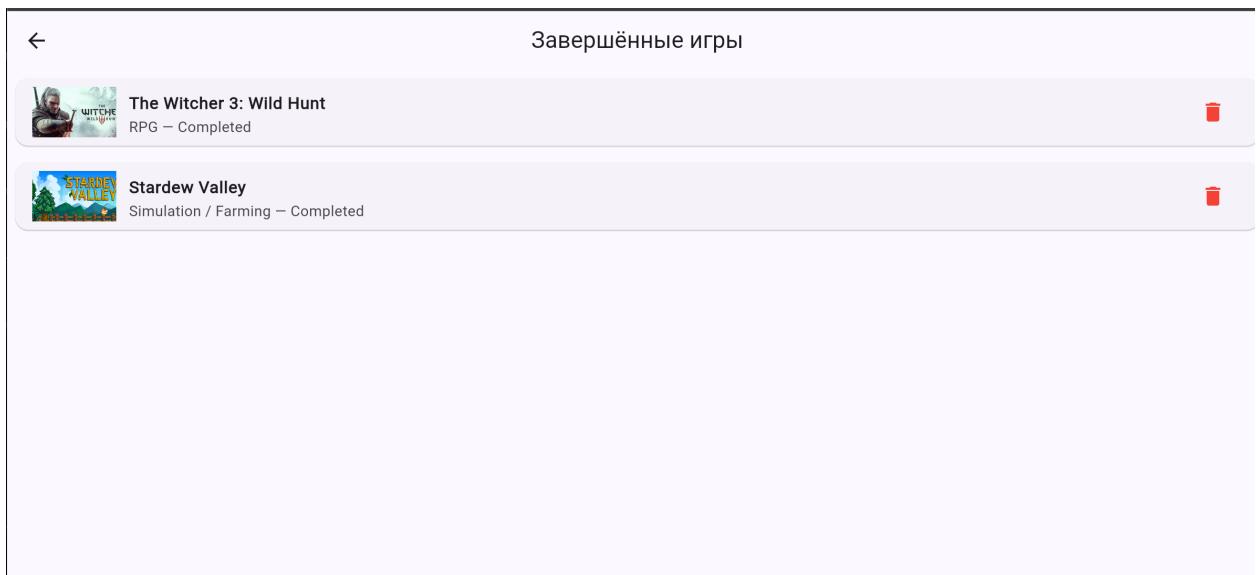


Рисунок 36 — Экран списка законченных игр до удаления выбранной карточки.

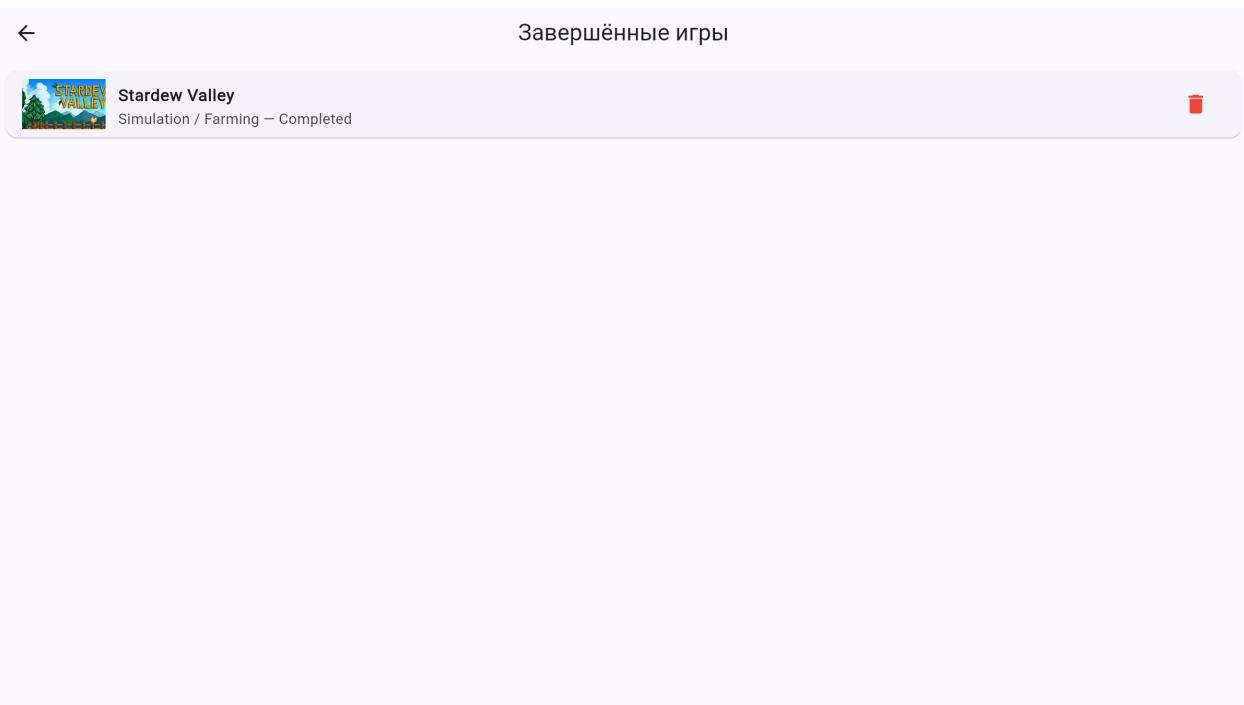


Рисунок 37 — Экран законченных игр после удаления.

В экране GameListScreen отрисовка списка игр реализована через доступ к общему состоянию приложения с использованием DI-контейнера GetIt. Непосредственно в методе build данныечитываются из зарегистрированного в контейнере сервиса GameService через обращение:

GetIt.I<GameService>().games (рисунок 38). После получения полного списка игр применяется локальная фильтрация по введённой пользователем строке поиска. Для этого используется контроллер текстового поля (TextEditingController), значение которого приводится к нижнему регистру, после чего выполняется фильтрация: `where((g) => g.title.toLowerCase().contains(query))`.

Таким образом формируется коллекция игр, соответствующая введённому запросу.

Полученный список передаётся в построение пользовательского интерфейса — формируется список карточек (GameCard), каждая из которых получает экземпляр модели Game и отображает информацию об игре: название, жанр, статус, изображение и рейтинг.

```
List<Game> _filteredGames() {  
    final repo = GetIt.I<GameService>();  
    final query = _searchController.text.toLowerCase();  
    return repo.games  
        .where((g) => g.title.toLowerCase().contains(query))  
        .toList();  
}
```

Рисунок 38 — Фрагмент метода build в GameListScreen.

Далее можем убедиться в корректности работы данной модификации посмотрев на состояния экранов на рисунках 39-40.

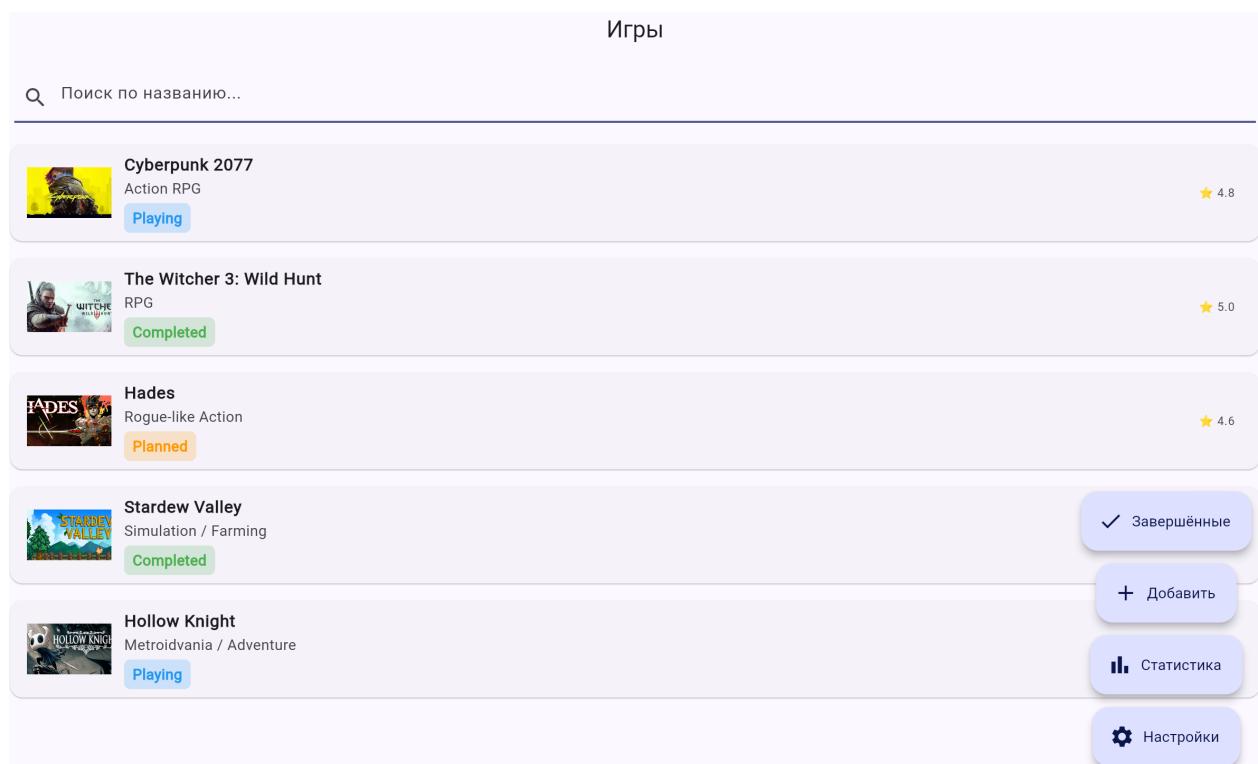


Рисунок 39 — Экран списка целей без фильтрации.

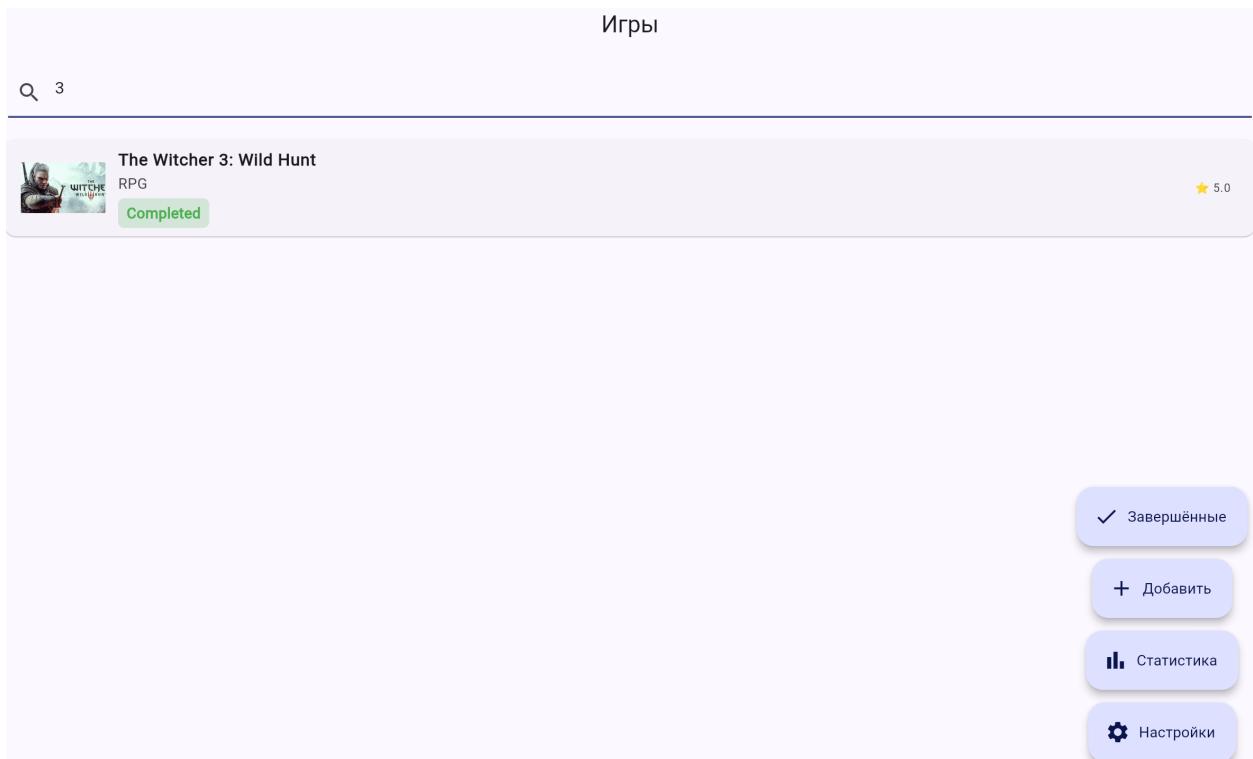


Рисунок 40 — Экран списка целей после применения поиска.

В экране AddEditGameScreen добавление новой игры реализовано через доступ к общему состоянию приложения с использованием DI-контейнера GetIt.

В методе `_save`, после заполнения пользователем всех необходимых полей, создаётся новый экземпляр модели `Game`, в который передаются значения, введённые на экране: название игры, жанр, статус, рейтинг, ссылка на изображение и комментарий (рисунок 41). После создания объекта вызов: `GetIt.I<GameService>().addGame(game);` добавляет сформированную игру в общий список `games`, которым управляет сервис `GameService`, зарегистрированный в контейнере GetIt..

```

void _save() {

    final game = Game(
        id: DateTime.now().millisecondsSinceEpoch.toString(),
        title: _titleController.text.trim(),
        genre: _genreController.text.trim(),
        status: _selectedStatus,
        imageUrl: _imageController.text.trim(),
        rating: double.tryParse(_ratingController.text) ?? 0,
        comment: _commentController.text.trim(),
    ); Game

    GetIt.I<GameService>().addGame(game);

    context.go("/");
}

```

Рисунок 41 — Фрагмент метода `_save` в `AddEditGameScreen` с вызовом `GetIt.I<GameService>(). addGame(game)`

Далее можем убедиться в корректности работы данной модификации, посмотрев на состояния экранов на рисунках 42-44.

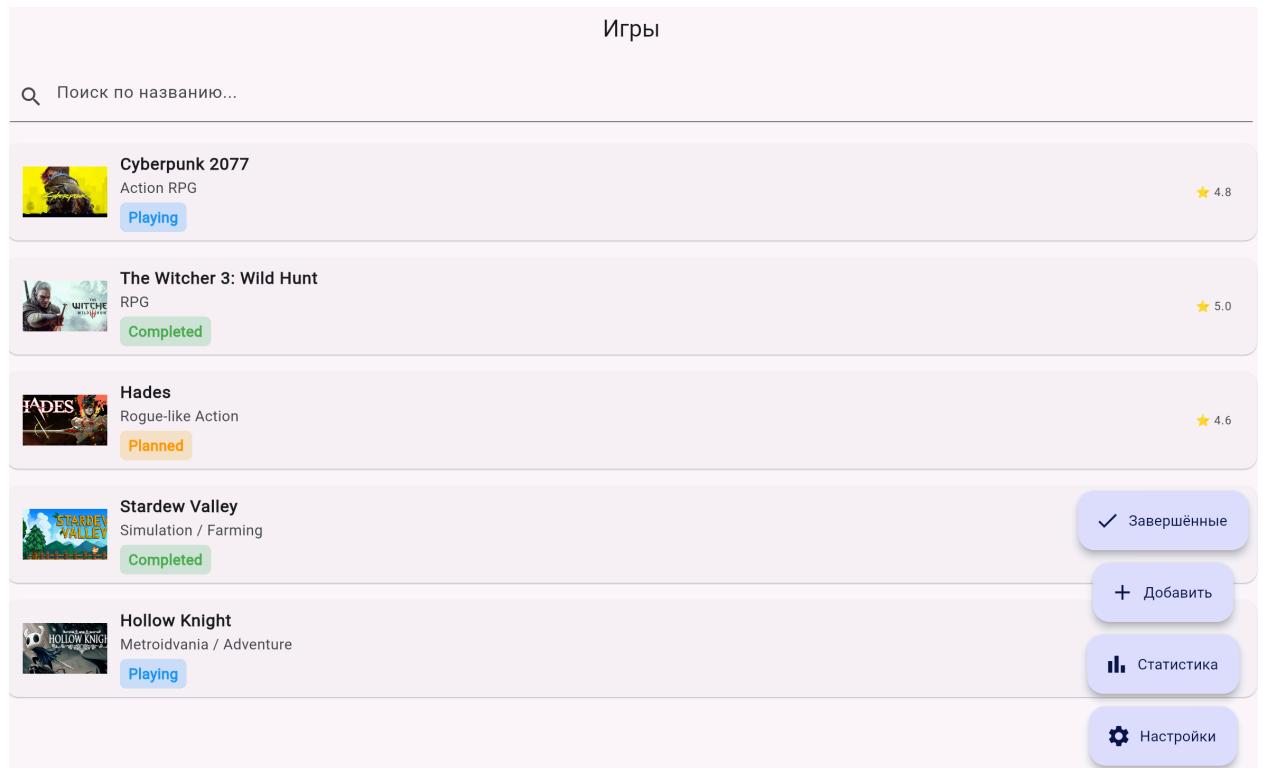


Рисунок 42 — Экран списка целей до добавления новой игры.

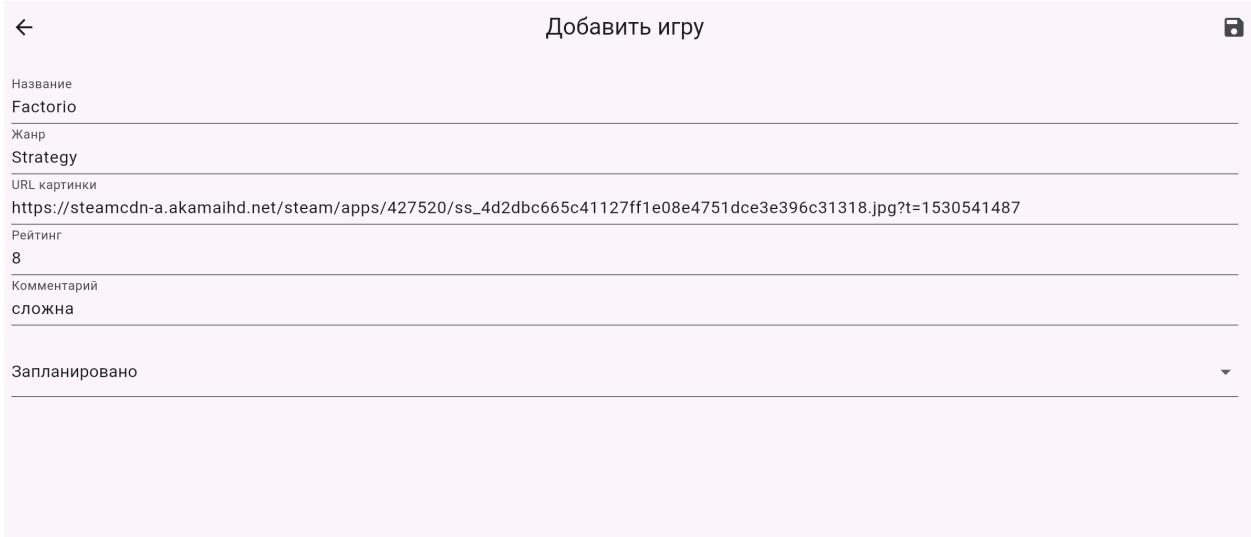


Рисунок 43 — Экран добавления игры с заполненными полями.

A screenshot of a game library list titled 'Игры' (Games). The list includes the following entries:

- The Witcher 3: Wild Hunt (Action RPG, Playing, 4.8 stars)
- Hades (Rogue-like Action, Planned, 4.6 stars)
- Stardew Valley (Simulation / Farming, Completed, 5.0 stars)
- Hollow Knight (Metroidvania / Adventure, Playing)
- Factorio (Strategy, Planned)

On the right side of the screen, there are four floating buttons with icons and text:

- ✓ Завершённые (Completed)
- + Добавить (Add)
- 📊 Статистика (Statistics)
- ⚙️ Настройки (Settings)

Рисунок 44 — Экран списка целей после добавления новой игры.

Заключение

Приложение успешно переработано для использования механизмов внедрения зависимостей с применением InheritedWidget (GameScope) и GetIt. Оба подхода позволили существенно улучшить архитектуру, повысив её читаемость, гибкость и расширяемость.

Использование InheritedWidget обеспечило тесную интеграцию со средой Flutter и оказалось удобным решением для небольших и средних приложений, где управление общим состоянием — например, списком игр — происходит непосредственно во виджетах. Такой подход позволил автоматически обновлять пользовательский интерфейс при изменении данных и упростил логику экрана благодаря реактивной перерисовке.

В свою очередь, интеграция контейнера зависимостей GetIt предоставила более универсальный и полностью независимый от контекста механизм доступа к сервису GameService. Это делает GetIt предпочтительным для проектов с выраженной модульностью и разделением на уровни данных, логики и UI. Получение зависимости без контекста упростило код экранов, позволило обращаться к состоянию из любого места приложения и повысило повторное использование логики.

Проведённое тестирование показало корректную работу всех функций приложения в обеих реализациях: игры успешно добавляются, редактируются, удаляются, фильтруются и изменяют статус; интерфейс своевременно отображает изменения без ошибок и задержек. Таким образом, оба варианта внедрения зависимостей доказали свою эффективность, а итоговая структура приложения стала более устойчивой и масштабируемой.

Все изменения, выполненные в результате данной практической работы, были сохранены в удаленном репозитории github:
https://github.com/RinaErxori/flutter_practice/tree/pr8