МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«УДМУРТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

(Филиал ФГБОУ ВО «УдГУ» в г. Воткинске)

ОТЧЕТ ПО ПРАКТИКЕ

Вид практики: производственная практика, преддипломная

(Вид производственной практики, ненужное убрать)

Специальность 09.02.07 Информационные системы и программирование

(Код, наименование специальности)

Профессиональный модуль: **нет**

Обучающегося 4 курса, группы СПО-09-Вт-090207-41(к)

Форма обучения \_\_\_\_\_\_\_\_очная\_\_\_\_\_\_\_\_\_\_

(очная, заочная)

Бобылев Роман Александрович

(Фамилия, имя, отчество)

Место практики Филиал ФГБОУ ВО «УдГУ» в г. Воткинске

(Название организации)

Срок практики с 05.05.2025 по 01.06.2025

Руководители практики

от организации \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

(должность) (подпись) (ФИО)

от филиала УдГУ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ Мамрыкин О. В.

(должность) (подпись) (ФИО)

Итоговая оценка по практике \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Воткинск 2025

Содержание

[Введение 3](#_Toc199681339)

[1 Доступные способы сохранения состояний во flame 4](#_Toc199681340)

[1.1 Использование SharedPreferences 4](#_Toc199681341)

[1.2 Сохранение данных в локальные файлы во Flutter (dart:io + path\_provider) 4](#_Toc199681342)

[1.3 Использование Hive (NoSQL-база данных) 7](#_Toc199681343)

[1.4 Сохранение в SQLite (через sqflite) 9](#_Toc199681344)

[1.5 Сохранение в облако (Firebase, REST API и др.) 12](#_Toc199681345)

[Заключение по разделу 12](#_Toc199681346)

[2 Реализация сохранения состояния с применением Hive 12](#_Toc199681347)

[2.1 Установка необходимых зависимостей 12](#_Toc199681348)

[2.2 Модификация класса «Игрока» для работы с Hive 13](#_Toc199681349)

[2.3 Инициализация хранилища и регистрация адаптера. 16](#_Toc199681350)

[2.4 Сохранение и загрузка состояния игрока с использованием хранилища Hive 17](#_Toc199681351)

[Заключение 19](#_Toc199681352)

[Список используемых источников 21](#_Toc199681353)

# Введение

В современной разработке мобильных и десктопных приложений на Flutter важную роль играет игровой движок Flame, который предоставляет удобные инструменты для создания 2D-игр. Одной из ключевых задач при разработке игр является сохранение и восстановление состояния игры, что позволяет пользователям продолжить игровой процесс с того же места после перезапуска приложения.

Актуальность данной темы обусловлена необходимостью обеспечения удобства пользователей и повышения стабильности работы приложений. Реализация механизмов сохранения и загрузки состояния в играх на Flutter Flame требует понимания работы движка, а также использования подходящих методов сериализации и хранения данных.

Целью данной практики является изучение и реализация механизмов сохранения и загрузки состояния в игровом движке Flame. В ходе работы предстоит рассмотреть следующие задачи:

- Изучение возможностей Flame для управления состоянием игры.

- Анализ методов сериализации данных (JSON, binary, Hive и др.).

- Реализация сохранения и восстановления игрового прогресса.

- Тестирование работоспособности решения на различных платформах.

Практическая значимость работы заключается в применении полученных знаний для создания устойчивого и удобного механизма сохранения состояния, что может быть использовано в дальнейшей разработке игровых проектов на Flutter.

# 1 Доступные способы сохранения состояний во flame

## 1.1 Использование SharedPreferences

SharedPreferences — это легковесное решение для хранения простых данных в формате ключ-значение. Он использует механизмы платформы (Android — SharedPreferences, iOS — NSUserDefaults), поэтому работает быстро и подходит для небольших данных (настройки, рекорды, флаги).

Как указано на странице самого расширения - обертывает платформу-специфическое постоянное хранилище для простых данных Данные могут сохраняться для диска асинхронно, и нет никакой гарантии, что записи будут сохраняться на диске после выполнения, поэтому этот плагин не должен использоваться для хранения критически важных данных.  Поддерживаемые типы: int, double, bool, String, List<String>.

Пример:

import 'package:shared\_preferences/shared\_preferences.dart';

// Сохранение

final prefs = await SharedPreferences.getInstance();

await prefs.setInt('highScore', 100);

// Загрузка

final highScore = prefs.getInt('highScore') ?? 0;

Несмотря на простоту использования, к сожалению не подходит для сохранения сложных структур данных(нужно сериализовать например в json). Не гарантирует сохранения данных при записи.

## 1.2 Сохранение данных в локальные файлы во Flutter (dart:io + path\_provider)

Этот метод подходит для хранения сложных данных (JSON, бинарные файлы, игровые уровни) в файловой системе устройства. В отличие от SharedPreferences, он позволяет работать с большими объемами данных и структурированными объектами.

Этот метод позволяет сохранять любые данные(json, csv, бинарные данные, изображения и аудио) в локальные файлы. Можно сохранять объекты предварительно обработанные dart:convert.

Пример:

Допустим, у нас есть объект Player:

class Player {

  final String name;

  final int level;

  final List<String> inventory;

  Player(this.name, this.level, this.inventory);

  // Преобразование в Map для JSON

  Map<String, dynamic> toJson() => {

'name': name,

'level': level,

'inventory': inventory,

  };

  // Создание из Map

  factory Player.fromJson(Map<String, dynamic> json) => Player(

json['name'],

json['level'],

List<String>.from(json['inventory']),

  );

}

Мы можем сохранить его в файл следующим образом:

import 'dart:io';

import 'dart:convert';

import 'package:path\_provider/path\_provider.dart';

Future<void> savePlayer(Player player) async {

  final directory = await getApplicationDocumentsDirectory();

  final file = File('${directory.path}/player.json');

  await file.writeAsString(jsonEncode(player.toJson()));

}

И восстановить из файла как показано ниже:

Future<Player?> loadPlayer() async {

  try {

final directory = await getApplicationDocumentsDirectory();

final file = File('${directory.path}/player.json');

if (!await file.exists()) return null;

final jsonData = await file.readAsString();

return Player.fromJson(jsonDecode(jsonData));

  } catch (e) {

print('Ошибка загрузки: $e');

return null;

  }

}

Использование этого способа позволяет сохранять любые данные, но есть определенные минусы, необходимо самостоятельно сериализовать данные.

## 1.3 Использование Hive (NoSQL-база данных)

Hive — это легковесная, быстрая и удобная NoSQL-база данных, написанная на чистом Dart. Она отлично подходит для хранения структурированных данных (игровые объекты, настройки, инвентарь) и работает быстрее SQLite и SharedPreferences в большинстве сценариев.

Hive хранит данные в Box - это аналог коллекций в NoSQL. Каждый Box может содержать: примитивы (int, String, bool, List, Map), кастомные объекты (если зарегистрирован адаптер). Данные хранятся в формате ключ → значение. Ключи могут быть int или String. Значения — любые поддерживаемые типы. Для работы с кастомными классами (например, Player, Item) нужно создать адаптер.

Пример:

Инициализация Hive:

import 'package:hive/hive.dart';

import 'package:hive\_flutter/hive\_flutter.dart';

void main() async {

  await Hive.initFlutter(); // Инициализация для Flutter

  await Hive.openBox('settings'); // Открываем Box

  runApp(MyApp());

}

Работа с примитивами:

final settingsBox = Hive.box('settings');

// Запись данных

settingsBox.put('soundEnabled', true);

settingsBox.put('playerName', 'Hero');

settingsBox.put('highScore', 1000);

// Чтение данных

final isSoundOn = settingsBox.get('soundEnabled', defaultValue: true);

final playerName = settingsBox.get('playerName', defaultValue: 'Guest');

final highScore = settingsBox.get('highScore', defaultValue: 0);

// Удаление данных

settingsBox.delete('highScore');

Пример работы с кастомными объектами:

Подключение адаптера для класса Player:

void main() async {

  await Hive.initFlutter();

  Hive.registerAdapter(PlayerAdapter()); // Регистрация адаптера

  await Hive.openBox<Player>('players'); // Box для Player

  runApp(MyApp());

}

Сохранение и загрузка объектов:

final playersBox = Hive.box<Player>('players');

// Сохранение объекта

playersBox.put('currentPlayer', Player('Hero', 10, ['Sword', 'Potion']));

// Чтение объекта

final player = playersBox.get('currentPlayer');

// Удаление

playersBox.delete('currentPlayer');

Данный метод сохранения данных подходит для задач сохранения информации о игровом процессе благодаря возможности сохранять кастомные классы через адаптеры. Но есть в нем и недостатки, это отсутствие поддержки какого-либо языка запросов.

## 1.4 Сохранение в SQLite (через sqflite)

**SQLite** - это встраиваемая **реляционная база данных**, которая подходит для сложных данных с отношениями (например, инвентарь, квесты, диалоги). Пакет sqflite предоставляет удобный интерфейс для работы с SQLite во Flutter.

Хорошо подходит для предметной области, выбранной для проекта – RPG. Можно описать структуру данных как несколько сущностей, в частности –персонаж(уровень, опыт и другие характеристики), предметы(список всех предметов в игре, советующие им имена файлов спрайтов и характеристики), инвентарь(отношение персонаж -> предметы) и тд. То есть этот подход данных можно использовать не только для сохранения текущего состояния игры, но и для удобного управления информацией об объектах игрового мира.

SQLite хорошо подходит для хранения игровой статистики, например можно сохранять информацию о том, когда и какого типа монстров побеждал игровой персонаж, эту статистику можно показывать на странице статистики по персонажу использую простой sql запрос.

Используя графический клиент для SQLite, разработчики могут удобно и наглядно редактировать наборы данных для игровых объектов.

Программный интерфейс sqflite так же достаточно прост.

Инициализация базы:

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

Future<Database> initDB() async {

final dbPath = await getDatabasesPath(); // Путь к папке с БД

final path = join(dbPath, 'game.db'); // Полный путь к файлу

return openDatabase(

path,

version: 1,

onCreate: (db, version) async {

// Создаём таблицы при первом запуске

await db.execute('''

CREATE TABLE players (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

level INTEGER DEFAULT 1,

gold INTEGER DEFAULT 0

)

''');

await db.execute('''

CREATE TABLE inventory (

id INTEGER PRIMARY KEY AUTOINCREMENT,

player\_id INTEGER,

item\_name TEXT,

FOREIGN KEY (player\_id) REFERENCES players(id)

)

''');

},

);

}

Работа с запросами:

Future<int> addPlayer(String name) async {

final db = await initDB();

return db.insert(

'players',

{'name': name, 'level': 1, 'gold': 0},

conflictAlgorithm: ConflictAlgorithm.replace,

);

}

Future<List<Map<String, dynamic>>> getPlayers() async {

final db = await initDB();

return db.query('players');

}

Future<void> deletePlayer(int id) async {

final db = await initDB();

await db.delete(

'players',

where: 'id = ?',

whereArgs: [id],

);

}

Future<void> updatePlayerLevel(int id, int newLevel) async {

final db = await initDB();

await db.update(

'players',

{'level': newLevel},

where: 'id = ?',

whereArgs: [id],

);

}

Однако у этого метода есть и свои недостатки. К сожалению, SQLite не отличается высокой производительностью, это связанно с тем, что, по сути, работа происходит с единственным файлом данных, расположенным на файловой системе устройства, в отличии от полноценных серверов баз данных таких как MySql, Postgresq и тд. Это негативно влияет на скорость выполнения операций чтения-записи. В том числе и потому, что SQLite не позволяет держать индексы в отдельной области памяти, как это реализовано в других СУБД. SQLite хранит индексы в основном файле базы данных, что может снижать производительность при больших объемах данных.

Есть минусы и у подхода, когда в проекте информация об игровых объектах в базе данных. В этом случаи невозможно отслеживать какие конкретно изменения в структуре игровых объектов были зафиксированных системой управления версий(git), потому что файл данных имеет бинарный формат.

## 1.5 Сохранение в облако (Firebase, REST API и др.)

В данном проекте не предусмотрены никаких сетевых функций, поэтому данный метод не подходит по определению.

# Заключение по разделу

Учитывая специфику приложения(игра RPG) наиболее подходящим вариантом является использования хранилища Hive, основными преимуществами которого является высокая производительность и удобный программный интерфейс.

# 2 Реализация сохранения состояния с применением Hive

## 2.1 Установка необходимых зависимостей

Для работы с Hive необходимо добавить в зависимости в файле pubspec.yaml:

dependencies:

hive: ^2.2.3

hive\_flutter: ^1.1.0

В зависимости кроме Hive добавлен и HiveFlutter, он используется для более простой интеграции с flutter и предоставляет методы инициализации хранилища в контексте flutter.

Для сохранения объектов необходимы специальные адаптеры, для их генерации необходимы следующие зависимости:

dev\_dependencies:

hive\_generator: ^2.0.1

build\_runner: ^2.4.6

Кроме самого генератора (hive\_generator) необходимо установить build\_runner. Пакет build\_runner предоставляет команды общего назначения для создания файлов, включая тестирование созданных файлов или обслуживание как исходных, так и сгенерированных файлов. Он необходим для вызова генератора hive\_generator из командной строки.

## 2.2 Модификация класса «Игрока» для работы с Hive

Перед тем как сохранить объект в хранилище Hive необходимо объявить HiveType, он используется для однозначной идентификации класса в хранилище Hive. Для этого необходимо перед объявлением класса указать специальную директиву как в примере ниже:

@HiveType(typeId: 0)

class PlayerComponent extends SpriteAnimationComponent

with CollisionCallbacks, HasGameRef<Game1> {

…

А также необходимо указать какие конкретно поля необходимо сохранить в хранилище Hive.

class PlayerComponent extends SpriteAnimationComponent

with CollisionCallbacks, HasGameRef<Game1> {

@HiveField(0) //Так мы объявляем, что это поле необходимо сохранить.

double attackSpeed = 1;

@HiveField(1)

double damage = 10;

@HiveField(2)

int experience = 0;

@HiveField(3)

int level = 1;

@HiveField(4)

double currentHealth = 200;

…

Для наших задач необходимо сохранять только некоторые поля класса PlayerComponent.

После того как определенны все поля, перед объявлением класс необходимо указать файл с классом адаптера, это специальный класс который будет сгенерирован автоматически. Имя подключаемого файла должно содержать имя текущего файла класса с расширением .g.dart.

part 'player-component.g.dart';

После того как эти подготовительные шаги завершены, можно запустить команду генерации адаптера. Для этого необходимо выполнить команду:

flutter pub run build\_runner build

Эту команду необходимо выполнять каждый раз после изменений связанных с метаданными Hive в классе. В результате выполнения этой команды будет сгенерирован файл player-component.g.dart. Если посмотреть на содержимое этого файла становиться понятно, что все данные раскладываются в хранилище последовательно, в зависимости от их типа:

class PlayerComponentAdapter extends TypeAdapter<PlayerComponent> {

@override

final int typeId = 0;

@override

PlayerComponent read(BinaryReader reader) {

final numOfFields = reader.readByte();

final fields = <int, dynamic>{

for (int i = 0; i < numOfFields; i++) reader.readByte(): reader.read(),

};

return PlayerComponent()

..atackSpeed = fields[0] as double

..damage = fields[1] as double

..experience = fields[2] as int

..level = fields[3] as int

..currentHealth = fields[4] as double;

}

@override

void write(BinaryWriter writer, PlayerComponent obj) {

writer

..writeByte(5)

..writeByte(0)

..write(obj.atackSpeed)

..writeByte(1)

..write(obj.damage)

..writeByte(2)

..write(obj.experience)

..writeByte(3)

..write(obj.level)

..writeByte(4)

..write(obj.currentHealth);

}

@override

int get hashCode => typeId.hashCode;

@override

bool operator ==(Object other) =>

identical(this, other) ||

other is PlayerComponentAdapter &&

runtimeType == other.runtimeType &&

typeId == other.typeId;

}

Так же становиться понятно, что данные из хранилища извлекаются и последовательно присваиваются заявленным полям. Таким образом Hive продолжает под капотом работать с примитивными типами данных.

## 2.3 Инициализация хранилища и регистрация адаптера.

Перед тем как использовать хранилище Hive во Flutter необходимо провести его инициализацию. Метод инициализации используется для инициализации библиотеки Hive перед началом её использования в приложении Flutter. Этот метод автоматически настраивает пути хранения данных, учитывая особенности платформ Android и iOS, что избавляет разработчика от ручного задания путей к файлам базы данных.

Для инициализации используется вызов:

await Hive.initFlutter();

После инициализации, необходимо зарегистрировать адаптер для работы с хранилищем.

Hive.registerAdapter(PlayerComponentAdapter());

Эти действия необходимо проделать один раз перед использованием хранилища. Повторная регистрация адаптера приведет к ошибке. Поэтому инициализацию и регистрацию адаптера лучше выполнять в самом начале, перед выполнением другого кода игры.

В текущем проекте инициализация и регистрация адаптера выполняется в теле функции main:

…

void main() async {

await Hive.initFlutter();

Hive.registerAdapter(PlayerComponentAdapter());

…

## 2.4 Сохранение и загрузка состояния игрока с использованием хранилища Hive

Для сохранения состояния игрока необходимо последовательно открыть хранилище и вызвать метод put. В текущем проекте это действие выполняется в классе PlayerComponent в методе update:

…

if (finalPosition != null && position.y < finalPosition!) {

var todoBox = await Hive.openBox<PlayerComponent>('player');

todoBox.put('player', this);

gameRef.router.pushReplacementNamed('camp');

}

…

Для загрузи информации о игроке в классе Game реализован метод loadPlayer он вызывается из главного меню в то время когда пользователь выбирает элемент «Продолжить».

loadPlayer() async {

// 1. Закрываем текущий мир

if (world != null) {

remove(world);

}

// 2. Загружаем сохранённого игрока из Hive

final playerBox = await Hive.openBox<PlayerComponent>('player');

if (playerBox.isNotEmpty && playerBox.get('player') != null) {

player = playerBox.get('player')!;

print("Игрок загружен из Hive: ${player.position}");

} else {

print("Сохранённый игрок не найден, создаём нового");

player = PlayerComponent(key: ComponentKey.named('player'));

}

// 3. Перезагружаем мир с новым игроком

router.pushReplacementNamed('main');

}

Таким образом реализуется загрузка и сохранение состояние игрока в процессе игры.

# Заключение

Эта практическая работа посвящена изучению и реализации механизмов сохранения и загрузки состояния игры в игровом движке Flame, используемом в приложениях на платформе Flutter. Основной целью практики было освоение подходов к сохранению и восстановлению игрового процесса, что позволило обеспечить удобство для пользователей и повысить стабильность работы приложения.

Основные выводы:

Рассмотрены различные подходы к сохранению данных, такие как использование SharedPreferences, локальных файлов, Hive, SQLite и облачных решений. Однако для конкретной задачи разработки игры RPG наиболее эффективным решением стало применение Hive.

Библиотека Hive продемонстрировала высокую производительность и удобное взаимодействие с кастомными структурами данных. Ее способность сохранять сложные структуры и поддержку быстрой обработки сделали выбор очевидным.

Практические аспекты, связанные с реализацией хранения данных с помощью Hive, включают создание специальных адаптеров для классов, упрощающих интеграцию с хранилищем. Процесс модификации существующих классов оказался простым и прозрачным.

Подчеркнута важность правильной инициализации и регистрации адаптера, а также последовательного выполнения операций сохранения и загрузки данных в рамках основного потока программы.

Таким образом, полученные знания и практические навыки позволят создавать устойчивые и удобные механизмы сохранения состояния в текущем проекте.

# Список используемых источников

1. Черников, Н. В. Разработка мобильных приложений на Flutter / Н. В. Черников. — СПб.: Питер, 2021. — 352 c.
2. Нечаев, Д. Ю. Основы мобильной разработки на Flutter : справочник и самоучитель / Д. Ю. Нечаев. — Москва : Эксмо, 2022. — 384 c.
3. Павлов, М. И. Создание мобильных приложений на Flutter : от нуля до профессионала / М. И. Павлов. — Москва : Манн, Иванов и Фербер, 2021. — 464 c.
4. Зубарев, Е. Г. Пошаговая разработка игр на Flutter / Е. Г. Зубарев. — Москва : Техносфера, 2022. — 304 c.