МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ САМАРСКОЙ ОБЛАСТИ ГАПОУ СО «САМАРСКИЙ ГОСУДАРСТВЕННЫЙ КОЛЛЕДЖ»

**КУРСОВАЯ РАБОТА**

по специальности: Информационные системы и программирование (специалист по информационным ресурсам)

по дисциплине «Разработка информационного контента (по отраслям)»

на тему **«Разработка системы для управления игровыми турнирами»**

**Выполнил:**

студент группы ИС-22-04

\_\_\_\_\_\_\_\_\_\_Ш. Ш. Тураев

**Преподователь:**

\_\_\_\_\_\_\_\_\_\_Р. Р. Ахтямов

Оценка «\_\_\_\_\_»

Самара, 2025 г.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc194858665)

[1. СИСТЕМОТЕХНИЧЕСКАЯ ЧАСТЬ 5](#_Toc194858666)

[1.1 Архитектура системы 5](#_Toc194858667)

[1.2 Выбор технологий 5](#_Toc194858668)

[1.3 Проектирование базы данных 7](#_Toc194858669)

[1.4 Описание API и его методов 11](#_Toc194858670)

[2. КОНСТРУКТОРСКО-ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ 13](#_Toc194858671)

[2.1 Разработка структуры бэкенда на платформе .NET 13](#_Toc194858672)

[2.2 Реализация регистрации пользователей 15](#_Toc194858673)

[2.3 Реализация функционала управления командами и турнирами 18](#_Toc194858674)

[2.4 Создание и управление турнирными сетками 20](#_Toc194858675)

[2.5 Обработка результатов матчей и подсчёт победителей 21](#_Toc194858676)

[ЗАКЛЮЧЕНИЕ 24](#_Toc194858677)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 26](#_Toc194858678)

[ПРИЛОЖЕНИЕ 27](#_Toc194858679)

# **ВВЕДЕНИЕ**

В последние годы онлайн-игры и киберспортивные турниры приобрели огромную популярность по всему миру. С развитием интернет-технологий появилась возможность организовывать игровые соревнования в различных форматах, объединяя игроков из разных уголков планеты. Эти турниры могут быть как любительскими, так и профессиональными, и для их успешной организации требуется эффективная система управления.

Целью данной курсовой работы является разработка системы для управления игровыми турнирами, которая будет обеспечивать функциональность для организации, проведения и отслеживания турниров. Основное внимание будет уделено созданию веб-API для управления турнирами, игроками и командами, а также автоматическому формированию турнирных сеток и учёту результатов матчей.

Проект включает в себя следующие основные задачи:

* Разработка архитектуры системы, включающей базы данных и необходимую логику для работы с игровыми турнирами.
* Разработка бэкенда на платформе .NET, включающего создание RESTful веб-API для взаимодействия с клиентскими приложениями.
* Реализация базовых функциональностей, таких как регистрация пользователей, создание и управление командами, организация турниров, создание турнирных сеток, а также подсчёт результатов матчей.
* Обеспечение возможности взаимодействия с клиентским приложением через API, что позволит гибко управлять процессом проведения турниров и упрощает добавление новых функциональных возможностей в будущем.

В ходе работы будет произведена оценка существующих систем управления турнирами, выделены их сильные и слабые стороны, а также предложены пути улучшения. В результате будет создано полнофункциональное решение для управления игровыми турнирами, доступное через API, которое обеспечит простоту и удобство в управлении турнирами для организаторов и участников.

Курсовая работа будет включать теоретическую часть, в которой будет рассмотрена концепция разработки API, а также практическую часть, в которой будет представлено описание структуры и реализация основного функционала системы. В качестве примера будет реализован прототип системы, позволяющий организовывать турниры, регистрировать участников и вычислять результаты матчей с помощью автоматической генерации турнирных сеток.

# **1. СИСТЕМОТЕХНИЧЕСКАЯ ЧАСТЬ**

## **1.1 Архитектура системы**

Архитектура системы управления игровыми турнирами представляет собой многоуровневую модель, включающую несколько ключевых компонентов. В основе системы лежит веб-API, разработанное с использованием фреймворка .NET, которое обеспечивает взаимодействие с базой данных и управляет процессом создания и проведения турниров [1][3].

На серверной стороне система использует архитектуру клиент-сервер, где сервер обрабатывает запросы от клиента, выполняет необходимую логику и взаимодействует с базой данных для хранения информации о пользователях, турнирах, командах и матчах. Для обеспечения работы системы используется Entity Framework, который позволяет работать с базой данных SQLite, обеспечивая безопасное и эффективное взаимодействие с данными[4].

Для обеспечения авторизации и аутентификации пользователей используются традиционные механизмы, такие как хранение паролей с использованием хеширования. Система позволяет пользователям создавать аккаунты, входить в систему и выполнять различные действия, такие как создание команд и участие в турнирах[5][10].

При реализации функционала управления турнирами, система позволяет создавать турниры, добавлять команды, генерировать сетки матчей и отслеживать результаты матчей. Алгоритм генерации сетки турнира обеспечивает корректное распределение команд и создание матчей в соответствии с их результатами.

## **1.2 Выбор технологий**

Для реализации веб-API системы управления игровыми турнирами был выбран стек технологий, который обеспечивает стабильность, масштабируемость и простоту разработки. Разработка системы ориентирована на создание RESTful API для взаимодействия с клиентскими приложениями и другими сервисами.

В качестве серверной платформы для создания веб-приложений было принято решение использовать .NET 8. Эта версия платформы предоставила всё необходимое для быстрого и удобного создания высокопроизводительных веб-приложений, включая полноценную поддержку RESTful API[1][2].

ASP.NET Core — это современный фреймворк для разработки веб-приложений и API, который работает на .NET 8. Он является кроссплатформенным, что позволяет разрабатывать и развертывать приложение на различных операционных системах, таких как Windows, Linux и macOS. Это даёт гибкость в случае расширения проекта и необходимости переноса на другую платформу в будущем.

ASP.NET Core обладает множеством встроенных возможностей, которые облегчают разработку. Например, система маршрутизации позволяет с лёгкостью обрабатывать HTTP-запросы, а также эффективно управлять зависимостями через контейнер внедрения зависимостей (Dependency Injection). Этот инструмент помогает разрабатывать чистый, легко тестируемый и поддерживаемый код.

Кроме того, ASP.NET Core поддерживает использование различных подходов к аутентификации и авторизации, таких как аутентификация через JWT, что позволяет гибко строить безопасность приложения. В рамках проекта, хотя использование JWT-токенов не предусмотрено, сам фреймворк предлагает массу возможностей для реализации таких механизмов.

В качестве системы управления базами данных была выбрана SQLite. Это лёгкая реляционная база данных, которая хранит все данные в одном файле. SQLite подходит для небольших и средних проектов, где не требуется сложная архитектура многосерверных баз данных. В данном случае она идеально подходит для решения задач по хранению данных о пользователях, турнирах, командах и матчах [4].

SQLite отличается простотой в настройке и минимальными требованиями к инфраструктуре. Это делает её отличным выбором для проектов, которые не требуют высокой нагрузки, но при этом нуждаются в надежности хранения данных и быстром доступе к ним. В контексте данной курсовой работы SQLite позволяет эффективно управлять небольшими объемами данных, такими как информация о турнирах, командах и их участниках.

Кроме того, SQLite предлагает широкий набор возможностей для работы с транзакциями и поддерживает стандарт SQL, что делает её удобной для разработчиков. Она обеспечивает высокую производительность на небольших данных и при этом проста в интеграции с приложением на .NET.

## **1.3 Проектирование базы данных**

Проектирование базы данных является ключевым этапом в разработке системы, так как от правильно спроектированной базы зависит эффективность работы всего приложения, его масштабируемость и надёжность. В рамках данной курсовой работы разрабатывается база данных, которая будет использоваться для хранения информации о пользователях, турнирах, командах и матчах. База данных должна поддерживать возможности для добавления новых записей, обновления данных, а также выполнения сложных запросов для получения статистики и другой информации [2].

Для данного проекта была выбрана реляционная база данных SQLite, которая идеально подходит для небольших и средних по масштабу приложений. Эта база данных обладает рядом преимуществ, таких как простота настройки, лёгкость в управлении и высокая производительность на небольших объёмах данных.

Реляционная база данных SQLite использует таблицы для хранения данных. Каждая таблица состоит из строк (записей) и столбцов (атрибутов). Для нашего проекта было определено несколько ключевых сущностей, каждая из которых будет представлена своей таблицей[12].

1. Пользователи: Эта таблица хранит данные о пользователях, таких как имя, электронная почта и пароль для аутентификации.
2. Турниры: Турниры являются основной сущностью в системе, которая определяет название турнира, его описание, дату начала и окончания, а также статус.
3. Команды: Каждая команда, участвующая в турнире, будет иметь свою запись в таблице команд. Каждая команда будет связана с одним или несколькими турнирами.
4. Матчи: Для каждого турнира необходимо организовать матчи. Матчи связываются с командами и имеют определённый статус, дату проведения и результаты.
5. Сетка турнира: Таблица сетки хранит информацию о раундах турнира и связанных с ними матчах. Каждый раунд связан с предыдущим раундом, что позволяет строить иерархию матчей в рамках турнира.
6. Участие команды в турнире: Эта таблица будет служить связующим звеном между командами и турнирами, в которых они участвуют.
7. Участие команд в матчах: Эта таблица используется для связи команд с конкретными матчами, отражая их участие в игре.

Таблица пользователей (Users): Таблица пользователей хранит информацию об игроках, которые регистрируются на платформе. Пользователи могут участвовать в турнирах, создавать команды и принимать участие в матчах.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | int | Уникальный идентификатор пользователя. |
| Email | string | Электронная почта пользователя, используется для аутентификации. |
| Password | string | Хэшированный пароль для входа пользователя в систему. |
| Name | string | Имя пользователя (никнейм). |

Таблица 1 ­– структура User

Таблица турниров хранит информацию о каждом турнире, включая название, дату начала и окончания, а также текущий статус турнира.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | Int | Уникальный идентификатор турнира. |
| Name | String | Название турнира. |
| Description | String | Описание турнира. |
| StartDate | DateTime | Дата и время начала турнира. |
| EndDate | DateTime | Дата и время окончания турнира. |
| Status | string | Статус турнира (например, "В процессе", "Завершён", "Отменён"). |

Таблица 2 – структура Tournament

Таблица команд хранит информацию о каждой команде, которая может участвовать в турнире. Каждая команда может быть связана с несколькими турнирами.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | int | Уникальный идентификатор команды. |
| Name | string | Название команды. |
| Tag | string | Сокращенное название команды. |
| Game | string | Название игры по которой создана команда. |
| CaptainId | int | Уникальный идентификатор создателя команды. |

Таблица 3 – структура Team

Таблица матчей хранит информацию о каждом проведённом матче в рамках турнира, включая его дату, статус и результаты.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | int | Уникальный идентификатор матча. |
| TournamentId | int | Идентификатор турнира, к которому относится матч. |
| MatchDate | DateTime | Дата и время проведения матча. |
| Status | string | Статус матча (например, "Не начат", "В процессе", "Завершён"). |
| ScoreTeamOne | int | Количество очков команды 1 в матче. |
| ScoreTeamTwo | int | Количество очков команды 2 в матче. |

Таблица 4 – структура Match

Таблица сетки турнира отвечает за создание и управление раундами турнира. Каждый раунд связан с предыдущим, что позволяет строить иерархию матчей.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | int | Уникальный идентификатор раунда. |
| TournamentId | int | Идентификатор турнира, к которому относится раунд. |
| Round | int | Номер раунда (например, 1-й раунд, полуфинал и т.д.). |
| MatchId | int | Идентификатор матча, соответствующего данному раунду. |
| WinnerRoundId | int | Идентификатор победителя раунда, который будет участвовать в следующем раунде. |

Таблица 5 – структура Bracket

Таблица участия команды в турнире служит для связи команд с турнирами, в которых они участвуют.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | INT | Уникальный идентификатор записи. |
| TeamId | INT | Идентификатор команды. |
| TournamentId | INT | Идентификатор турнира, в котором команда участвует. |

Таблица 6 - структура таблицы TeamInTournament

Таблица участия команды в матче используется для связи команд с матчами, в которых они участвуют.

|  |  |  |
| --- | --- | --- |
|  | Тип данных | Описание |
| Id | INT | Уникальный идентификатор записи. |
| TeamOneId | INT | Идентификатор первой команды, участвующей в матче. |
| TeamTwoId | INT | Идентификатор второй команды, участвующей в матче. |
| MatchId | INT | Идентификатор матча. |

Таблица 7 - Структура таблицы TeamInMatch

Для обеспечения целостности данных между таблицами в базе данных используются внешние ключи, которые устанавливают связи между различными сущностями. Пример таких связей:

* В таблице Matches внешний ключ TournamentId связывает матч с турниром;
* В таблице Brackets внешний ключ TournamentId связывает раунд с турниром, а MatchId указывает на матч, который соответствует этому раунду;
* В таблице TeamInTournament внешний ключ TeamId связывает команду с турниром;
* В таблице TeamInMatch внешний ключ MatchId связывает команду с матчем.

При проектировании базы данных использовался принцип нормализации, чтобы минимизировать избыточность данных и избежать аномалий при добавлении, удалении и обновлении информации. На основании нормализации данные в таблицах организованы так, чтобы каждая таблица отвечала за одну сущность, и все связи между таблицами осуществляются через идентификаторы.

## **1.4 Описание API и его методов**

API будет работать по принципу REST, используя стандартные HTTP методы (GET, POST, PUT, DELETE) для выполнения операций с сущностями базы данных. Все данные, передаваемые между клиентом и сервером, будут сериализованы в формат JSON. Каждая операция будет иметь соответствующий маршрут (endpoint) и обеспечивать работу с нужной сущностью системы [3][5][7].

* GET — используется для получения данных с сервера.
* POST — используется для создания новых записей в базе данных.
* PUT — используется для обновления существующих записей.
* DELETE — используется для удаления записей.

API будет включать аутентификацию и авторизацию, обеспечивая доступ к определенным ресурсам только зарегистрированным пользователям.

|  |  |  |
| --- | --- | --- |
| Метод | Путь | Описание |
| POST | /api/auth/register | Метод для регистрации нового пользователя. Проверяет, существует ли уже пользователь с таким email, и если нет, создаёт нового пользователя. |
| POST | /api/auth/login | Метод для входа пользователя. Проверяет существование пользователя с данным email и совпадение пароля. Если всё правильно, возвращает информацию о пользователе. |
| GET | /api/team/all | Получить список всех команд. |
| GET | /api/team/{teamId}/roster | Получить состав команды по её ID. |
| POST | /api/team/create | Создать новую команду. |
| POST | /api/team/{teamId}/add-user | Добавить пользователя в команду. |
| DELETE | /api/team/remove-user/{userId} | Удалить пользователя из команды. |
| DELETE | /api/team/remove/{teamId} | Удалить команду. |
| POST | /api/tournament | Создание нового турнира. |
| GET | /api/tournament | Получить список всех турниров. |
| GET | /api/tournament/{id} | Получить информацию о турнире по его ID. |
| GET | /api/tournament/{tournamentId}/members | Получить участников турнира по его ID. |
| POST | /api/tournament/{tournamentId}/add-team/{teamId} | Добавить команду в турнир. |
| DELETE | /api/tournament/{tournamentId}/remove-team/{teamId} | Удалить команду из турнира. |
| POST | /api/tournament/{tournamentId}/generate-bracket | Генерация турнирной сетки для турнира. |
| POST | /api/tournament/{matchId}/set-result | Установка результата матча турнира. |
| GET | /api/user/all | Получить список всех пользователей. |
| PUT | /api/user/update/{id} | Обновить данные пользователя по его ID. |

Таблица 8 – методы API turaevApp

# **2. КОНСТРУКТОРСКО-ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ**

## **2.1 Разработка структуры бэкенда на платформе .NET**

Бэкенд системы управления игровыми турнирами был реализован на платформе .NET 8 с применением шаблона Web API. В качестве основы была выбрана многослойная архитектура, которая предполагает чёткое разделение компонентов по их назначению. Серверная часть проекта состоит из нескольких логических слоёв: контроллеров, моделей, DTO-объектов, контекста базы данных и миграций. Также реализована базовая бизнес-логика, которая отвечает за генерацию турниров, обработку матчей, добавление и удаление команд, а также регистрацию пользователей.

Контроллеры выполняют роль промежуточного звена между клиентом и внутренней логикой приложения. Каждый контроллер отвечает за отдельную сущность системы [7]. Например, TournamentController управляет созданием турниров, генерацией турнирной сетки и добавлением команд в турниры. TeamController отвечает за управление командами и их участниками. AuthController реализует регистрацию и вход пользователей. UserController позволяет просматривать и обновлять информацию о зарегистрированных пользователях.

Работа с базой данных осуществляется при помощи технологии Entity Framework Core, которая предоставляет удобный ORM-слой [4]. Вся логика взаимодействия с базой инкапсулируется в классе TuraevDbContext, который унаследован от DbContext. Внутри этого класса определены наборы данных, соответствующие таблицам базы: пользователи, команды, турниры, матчи, связи между пользователями и командами, между командами и турнирами, а также информация о матчах и турнирной сетке.

Для упрощения передачи данных между клиентом и сервером, а также для валидации входящей информации, были созданы специальные DTO-классы. Например, TournamentCreateDto используется для создания турнира, MatchResultDto — для передачи результата матча, UserUpdateDto — для обновления профиля пользователя. Это позволяет не передавать лишнюю информацию и не раскрывать внутреннюю структуру моделей клиенту.

Контекст базы данных TuraevDbContext конфигурируется в точке входа приложения, а именно в файле Program.cs. Здесь же настраивается использование базы данных SQLite, которая была выбрана как простое и удобное решение для учебного проекта. Работа с миграциями осуществляется через встроенные команды Entity Framework. С их помощью создаётся структура базы данных, а при необходимости можно вносить изменения в таблицы без потери данных.

Пример регистрации нового пользователя можно увидеть в следующем методе контроллера:

[HttpPost("register")]

public async Task<IActionResult> Register([FromBody] UserDto request)

{

if (await \_context.Users.AnyAsync(u => u.Email == request.Email))

return BadRequest(new { message = "Email уже используется" });

var hashedPassword = HashPassword(request.Password);

var user = new User { Name = request.Name, Email = request.Email, Password = hashedPassword };

\_context.Users.Add(user);

await \_context.SaveChangesAsync();

return Ok(new { message = "Регистрация успешна!" });

}

В данном фрагменте реализована логика проверки уникальности email, хеширования пароля и сохранения нового пользователя в базе. Это один из типичных примеров взаимодействия контроллера с базой данных через контекст.

Бизнес-логика, такая как генерация турнирной сетки, была реализована непосредственно в контроллере. Например, при генерации сетки команд происходит формирование матчей и создание записей в таблицах Matches, TeamsInMatches и Brackets. Каждый матч связывается с двумя командами, при этом, если количество команд нечётное, последняя команда может перейти в следующий раунд без соперника. После завершения каждого раунда система проверяет, остался ли только один победитель. Если да — турнир считается завершённым, и победитель сохраняется в таблицу Tournaments.

Весь этот процесс осуществляется пошагово с сохранением изменений после каждого этапа. Такой подход, хотя и упрощён, наглядно демонстрирует механику организации спортивных соревнований и отражает практическое применение концепций многослойной архитектуры и объектно-ориентированного проектирования в рамках .NET.

Таким образом, структура бэкенда построена логично, модульно и масштабируемо. Несмотря на то, что в рамках проекта отсутствует выделенный сервисный слой и система авторизации на основе токенов, текущая реализация обеспечивает стабильную и понятную архитектуру, подходящую для учебной цели.

## **2.2 Реализация регистрации пользователей**

Регистрация пользователей — один из ключевых компонентов любой информационной системы, связанной с управлением данными и доступом. В рамках системы управления игровыми турнирами была реализована базовая регистрация новых пользователей с сохранением их данных в базе и защитой паролей путём хеширования. Данный функционал был реализован в контроллере AuthController, созданном специально для работы с регистрацией и авторизацией.

Процесс регистрации построен следующим образом. Клиент отправляет POST-запрос по маршруту /api/auth/register, передавая в теле запроса данные нового пользователя: имя, адрес электронной почты и пароль. На сервере эти данные принимаются в виде объекта UserDto. Перед созданием нового пользователя производится проверка, существует ли уже пользователь с таким email. Это необходимо для предотвращения повторной регистрации с одинаковыми учётными данными.

Если email свободен, система переходит к этапу хеширования пароля. Пароль пользователя никогда не сохраняется в открытом виде — вместо этого используется метод HashPassword, реализующий безопасное одностороннее шифрование. Для упрощения проекта был выбран алгоритм SHA256, но в реальных системах предпочтительнее использовать более защищённые механизмы, такие как bcrypt или Argon2.

После хеширования пароля создаётся новый объект User, который сохраняется в базу данных через контекст TuraevDbContext. По завершении операции сервер возвращает сообщение об успешной регистрации. Пример кода, реализующего эту логику, приведён ниже:

[HttpPost("register")]

public async Task<IActionResult> Register([FromBody] UserDto request)

{

if (await \_context.Users.AnyAsync(u => u.Email == request.Email))

return BadRequest(new { message = "Email уже используется" });

var hashedPassword = HashPassword(request.Password);

var user = new User

{

Name = request.Name,

Email = request.Email,

Password = hashedPassword

};

\_context.Users.Add(user);

await \_context.SaveChangesAsync();

return Ok(new { message = "Регистрация успешна!" });

}

Метод HashPassword, вызываемый внутри контроллера, реализован с использованием стандартных средств шифрования .NET и работает следующим образом:

private string HashPassword(string password)

{

using var sha256 = SHA256.Create();

var bytes = Encoding.UTF8.GetBytes(password);

var hash = sha256.ComputeHash(bytes);

return Convert.ToBase64String(hash);

}

Этот метод преобразует пароль в массив байтов, шифрует его с помощью SHA256 и возвращает строку в формате Base64. Таким образом, даже если злоумышленник получит доступ к базе данных, он не сможет легко восстановить оригинальные пароли пользователей [11][7].

Также в будущем можно будет реализовать авторизацию, в том числе с использованием токенов (например, JWT), что обеспечит аутентификацию и защиту API от несанкционированного доступа. Однако в рамках данной курсовой авторизация не реализуется, так как основной акцент сделан на архитектуре и работе серверной части приложения.

Регистрация пользователей организована просто, но надёжно. Она демонстрирует базовые принципы работы с пользовательскими данными, защиту чувствительной информации и взаимодействие клиента с сервером через REST API. Такой подход соответствует требованиям учебного проекта и может быть легко расширен в будущем для поддержки дополнительных функций, таких как подтверждение email, восстановление пароля или привязка к социальным сетям.

## **2.3 Реализация функционала управления командами и турнирами**

В рамках разработанного веб-приложения важной частью функциональности является возможность управления командами и турнирами. Эти модули тесно связаны друг с другом: команды регистрируются, затем могут участвовать в турнирах, а турнирная система в свою очередь отслеживает их прогресс и формирует сетку матчей. Реализация этого функционала охватывает создание, удаление, редактирование и взаимодействие между командами и турнирами через REST API.

Управление командами реализовано в контроллере TeamController. Система позволяет создавать команды, добавлять в них участников, удалять участников и сами команды. При создании команды пользователь указывает название, тэг (сокращённое имя), название игры и ID капитана. Эти данные поступают на сервер в виде DTO-модели CreateTeamDto, а затем сохраняются в таблицу Teams. После успешного создания команды капитан автоматически добавляется в неё как участник — запись об этом создаётся в связующей таблице UsersInTeams, которая реализует связь многие-ко-многим между пользователями и командами.

Чтобы добавить участника в команду, клиент отправляет POST-запрос с идентификатором пользователя и команды. Контроллер проверяет, существует ли такая команда и пользователь, а также не состоит ли пользователь уже в другой команде. Если все проверки пройдены, создаётся новая запись в таблице UsersInTeams, связывающая пользователя с выбранной командой.

Удаление пользователя из команды реализовано через метод, который ищет соответствующую запись в связующей таблице и, если она существует, удаляет её. Удаление команды возможно только в том случае, если она пуста — то есть, в ней не осталось ни одного участника. Это необходимо для предотвращения случайной потери данных, связанных с участниками.

Параллельно с этим разрабатывался функционал для работы с турнирами, размещённый в контроллере TournamentController. Система позволяет создавать турниры с указанием названия, описания и даты начала. Каждый турнир может содержать несколько команд-участников, которые добавляются по аналогичной логике через отдельную сущность TeamsInTournaments.

Для добавления команды в турнир производится проверка наличия самого турнира и команды, а также удостоверение, что команда ещё не участвует в данном турнире. Если все условия выполнены, создаётся новая запись в таблице TeamsInTournaments, обозначающая участие данной команды в выбранном турнире. Удаление команды из турнира происходит аналогичным образом — соответствующая запись удаляется из базы[6].

После набора участников возможна генерация турнирной сетки. Для этого предусмотрен отдельный метод, формирующий пары команд и создающий матчи первого раунда. Каждая пара сохраняется как отдельный матч в таблице Matches, и дополнительно создаётся запись в таблице Brackets, отражающая структуру турнира. Каждый матч содержит две команды, дату проведения, статус, а также счёт, который обновляется по завершении встречи.

Когда результаты матчей обновляются, система проверяет, завершён ли раунд. Если все матчи раунда сыграны, генерируется следующий раунд, включающий только победителей. Таким образом, турнирная сетка формируется автоматически, а финальный победитель определяется на основе результатов матчей последнего раунда. При завершении турнира в таблице Tournaments сохраняется информация о победителе и дате завершения.

Функционал управления командами и турнирами демонстрирует логическую связанность сущностей, работу с реляционной базой данных через Entity Framework Core и применение принципов REST для построения API. Вся бизнес-логика размещена в контроллерах, и при необходимости она может быть вынесена в отдельные сервисы для повышения масштабируемости и читаемости кода.

## **2.4 Создание и управление турнирными сетками**

Одним из ключевых элементов функционала разработанной системы является возможность автоматической генерации и управления турнирной сеткой. Эта часть реализована в контроллере TournamentController и обеспечивает последовательное формирование матчей между командами, определение победителей, переход к следующим раундам, а также финальное завершение турнира с фиксацией победителя [5].

Создание турнирной сетки начинается после того, как в турнир добавлено достаточное количество команд. Минимальное число участников, необходимое для генерации сетки — две команды. Если это условие не выполнено, система возвращает соответствующую ошибку. В противном случае запускается логика генерации первого раунда [12].

На первом этапе из таблицы TeamsInTournaments извлекаются все команды, участвующие в выбранном турнире. Затем они попарно разбиваются на пары: каждая пара становится отдельным матчем. Если общее количество команд нечётное, последняя команда остаётся без пары и автоматически проходит в следующий раунд.

Для каждой пары создаётся объект Match, в котором указываются ID турнира, дата матча (например, текущая дата плюс номер раунда), статус (по умолчанию — Scheduled), а также поля счёта, которые изначально равны нулю. После этого создаётся соответствующая запись в таблице TeamInMatch, где указываются идентификаторы обеих команд-участниц.

Дополнительно создаётся запись в таблице Bracket, отражающая положение матча в турнирной сетке. В этой таблице фиксируются номер раунда, ID матча, а также поле WinnerRoundId, в которое будет записан победитель встречи после завершения. Таким образом, таблица Brackets содержит логику перехода между раундами и позволяет отследить структуру турнира.

Когда матч завершается и в него вносятся результаты, система определяет победителя путём сравнения счёта. В зависимости от победителя обновляется статус матча и записывается идентификатор выигравшей команды в таблицу Brackets. После завершения всех матчей текущего раунда происходит автоматическая генерация следующего раунда, в котором участвуют только победители предыдущих встреч. Новый раунд формируется по аналогии с первым, вплоть до финального матча.

Когда в турнире остаётся один победитель, система сохраняет его ID в таблице Tournaments, устанавливает дату окончания турнира и меняет статус турнира на завершённый. Таким образом, весь процесс — от первого раунда до финала — происходит автоматически, без необходимости вмешательства администратора.

Такой подход обеспечивает прозрачность и масштабируемость: система может обрабатывать любое количество команд и корректно строить сетку как на 4, так и на 32 участника. Архитектура решения позволяет при необходимости внедрить поддержку более сложных форматов турниров — например, двойное выбывание, групповые этапы или рейтинговую систему.

Весь механизм турнирной сетки демонстрирует применение принципов автоматизации в игровой логике, умение работать с реляционными структурами и последовательной бизнес-логикой, а также глубокую интеграцию между сущностями базы данных и логикой API. За счёт этого реализована полная поддержка жизненного цикла турнира от создания до подведения итогов.

## **2.5 Обработка результатов матчей и подсчёт победителей**

После генерации турнирной сетки и назначения матчей, ключевым этапом становится корректная обработка результатов сыгранных встреч. Этот функционал отвечает за фиксацию исходов матчей, определение победителей и построение последующих раундов на основании полученных данных. Он реализован в методе SetMatchResult контроллера TournamentController.

Каждый матч хранится в таблице Matches, где содержится информация о счёте команд (ScoreTeamOne, ScoreTeamTwo), дате проведения и статусе матча (Scheduled, Completed). Для обработки результатов администратор (или ответственный пользователь) вызывает метод API, передавая идентификаторы команд и итоговый счёт [13].

При получении запроса система находит соответствующий матч по его ID. Если матч не найден, возвращается ошибка. В противном случае происходит обновление счёта и установка статуса матча как Completed, сигнализируя о его завершении.

Затем система вычисляет победителя: сравниваются показатели счёта двух команд, и определяется та, которая набрала больше очков. Идентификатор победившей команды сохраняется в таблице Brackets в поле WinnerRoundId. Это поле играет ключевую роль в переходе команды в следующий раунд — только команды, победившие в своих матчах, участвуют в следующем этапе.

После обновления информации о победителе система проверяет, завершены ли все матчи текущего раунда. Для этого из таблицы Brackets извлекаются все записи, относящиеся к текущему раунду, и проверяется, заполнены ли у них все поля WinnerRoundId. Если все матчи завершены, формируется следующий раунд: победители предыдущих матчей попарно объединяются в новые встречи. Для каждой пары создаются новые записи в таблицах Matches, TeamInMatch и Brackets, как и при генерации первого раунда.

Процесс повторяется до тех пор, пока в турнире не останется один победитель. Если после завершения очередного раунда система обнаруживает, что количество победителей — один, это означает окончание турнира. В таблице Tournaments сохраняется идентификатор победившей команды (WinnerId) и устанавливается дата окончания турнира (EndDate), после чего турнир считается завершённым.

Подобная автоматизация исключает необходимость ручного администрирования хода турнира и позволяет системе полностью самостоятельно вести учёт матчей, побед и продвижения команд. Это критически важно для надёжной работы турнирной системы, особенно при большом числе участников и раундов.

Благодаря такой реализации достигается высокая степень согласованности данных, исключаются логические ошибки в расчётах, а также обеспечивается прозрачность — на любом этапе можно отследить статус матчей, определить текущих победителей и просмотреть прогресс турнира.

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсовой работы была разработана и реализована серверная часть информационной системы для управления игровыми турнирами. Основной целью проекта являлось создание функционального веб-API, обеспечивающего регистрацию пользователей, формирование команд, организацию турниров и автоматическое построение турнирных сеток с последующей фиксацией результатов матчей. Все ключевые задачи были успешно реализованы на платформе .NET с использованием технологии ASP.NET Core.

В процессе разработки были подробно изучены и применены принципы построения RESTful API, организации реляционных связей между сущностями в базе данных, а также подходы к разработке отказоустойчивой и масштабируемой архитектуры. Особое внимание было уделено логике турнирной системы: от регистрации команд до автоматического продвижения победителей по турнирной сетке.

Для хранения и управления данными использовалась база данных SQLite, что позволило обеспечить быструю настройку и лёгкость развёртывания системы в локальных условиях. Взаимодействие между сущностями реализовано через Entity Framework Core, что упростило процесс работы с данными и обеспечило надёжность при выполнении запросов.

Система обеспечивает стабильную и предсказуемую работу всех ключевых модулей: создание и редактирование пользователей, управление командами и турнирами, автоматическую генерацию матчей, а также обработку результатов вплоть до определения победителя турнира. Проект обладает логически чёткой архитектурой и может быть расширен дополнительным функционалом, таким как авторизация, интеграция с фронтендом, публикация турниров в открытом доступе и аналитика по матчам.

Разработка системы позволила не только закрепить практические навыки работы с .NET и ASP.NET Core, но и получить ценный опыт в проектировании API, работе с базами данных, а также в организации бизнес-логики приложений. Полученные знания и подходы могут быть использованы при создании более масштабных и сложных веб-приложений в будущем.

Таким образом, поставленные в начале работы цели были достигнуты в полном объёме, а результаты разработки соответствуют функциональным требованиям, заявленным в рамках темы курсового проекта.

# **СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Microsoft. ASP.NET Core в .NET 8. Документация [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-8.0.
2. Esposito D. Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure. Microsoft Press, 2022.
3. Freeman A. Pro ASP.NET Core 8 MVC. Apress, 2024. – 1220 с.
4. Entity Framework Core Documentation [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/ef/core/>
5. Microsoft. Web API в ASP.NET Core [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/aspnet/core/web-api/?view=aspnetcore-8.0>
6. Юмашев С.В. ASP.NET Core. Разработка приложений для Интернета. СПб.: Питер, 2023. – 512 с.
7. Bogard J. API Design in .NET. Leanpub, 2023. – Режим доступа: https://leanpub.com/apidesign
8. Bashmakov S.A. Безопасность веб-приложений: подходы, технологии, инструменты. – М.: БХВ-Петербург, 2022. – 356 с.
9. Рихтер Д. CLR via C#. 5-е изд. – М.: Вильямс, 2023. – 896 с.
10. Galloway J., Wilson P., Allen D. Professional ASP.NET MVC 5. Wrox, 2020.
11. Syme D. Expert F# 5.0: With .NET Core. Apress, 2021.
12. Pialorsi P. Programming Microsoft LINQ in .NET Framework. Microsoft Press, 2020.
13. Головачев А.В. Современное программирование на C#. – М.: ДМК Пресс, 2022. – 400 с.

# **ПРИЛОЖЕНИЕ**

**Листинг кода**

**Program.cs**

using Microsoft.EntityFrameworkCore;

using Turaev.WebAPI.Data;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<TuraevDbContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddCors(options =>

{

options.AddPolicy("AllowAllOrigins", builder =>

{

builder.AllowAnyOrigin()

.AllowAnyMethod()

.AllowAnyHeader();

});

});

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.UseCors("AllowAllOrigins");

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

**TuraevDbContext.cs**

using Microsoft.EntityFrameworkCore;

using Turaev.WebAPI.Models;

namespace Turaev.WebAPI.Data

{

public class TuraevDbContext : DbContext

{

public TuraevDbContext(DbContextOptions<TuraevDbContext> options) : base(options) { }

public DbSet<User> Users { get; set; }

public DbSet<Team> Teams { get; set; }

public DbSet<Tournament> Tournaments { get; set; }

public DbSet<Match> Matches { get; set; }

public DbSet<UserInTeam> UsersInTeams { get; set; }

public DbSet<TeamInMatch> TeamsInMatches { get; set; }

public DbSet<TeamInTournament> TeamsInTournamnets { get; set; }

public DbSet<Bracket> Brackets { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=turaevdatabase.db");

}

}

}

**AuthController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Security.Cryptography;

using System.Text;

using Turaev.WebAPI.Data;

using Turaev.WebAPI.Models;

namespace Turaev.WebAPI.Controllers

{

[Route("api/auth")]

[ApiController]

public class AuthController : ControllerBase

{

private readonly TuraevDbContext \_context;

public AuthController(TuraevDbContext context)

{

\_context = context;

}

[HttpPost("register")]

public async Task<IActionResult> Register([FromBody] UserDto request)

{

if (await \_context.Users.AnyAsync(u => u.Email == request.Email))

return BadRequest(new { message = "Email уже используется"});

var hashedPassword = HashPassword(request.Password);

var user = new User { Name = request.Name, Email = request.Email, Password = hashedPassword };

\_context.Users.Add(user);

await \_context.SaveChangesAsync();

return Ok(new { message = "Регистрация успешна!" });

}

[HttpPost("login")]

public async Task<IActionResult> Login([FromBody] UserDto request)

{

var user = await \_context.Users.FirstOrDefaultAsync(u => u.Email == request.Email);

if (user == null || !VerifyPassword(request.Password, user.Password))

return Unauthorized(new { message = "Неверный email или пароль" });

return Ok(user);

}

private static string HashPassword(string password)

{

using var sha256 = SHA256.Create();

return Convert.ToBase64String(sha256.ComputeHash(Encoding.UTF8.GetBytes(password)));

}

private static bool VerifyPassword(string inputPassword, string storedPassword)

{

return HashPassword(inputPassword) == storedPassword;

}

}

}

**TeamController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Xml;

using Turaev.WebAPI.Data;

using Turaev.WebAPI.Models;

namespace Turaev.WebAPI.Controllers

{

[Route("api/team")]

[ApiController]

public class TeamController : ControllerBase

{

private readonly TuraevDbContext \_context;

public TeamController(TuraevDbContext context)

{

\_context = context;

}

[HttpGet("all")]

public ActionResult<IEnumerable<Team>> GetAllTeam()

{

return Ok(\_context.Teams.ToList());

}

[HttpGet("{teamId}/roster")]

public ActionResult<IEnumerable<UserInTeam>> GetRosterTeam(int teamId)

{

return Ok(\_context.UsersInTeams.Where(t => t.TeamId == teamId).ToList());

}

[HttpPost("create")]

public async Task<IActionResult> CreateTeam([FromBody] CreateTeamDto dto)

{

var team = new Team { Name = dto.Name, Tag = dto.Tag, Game = dto.Game, CaptainId = dto.CaptainId };

\_context.Teams.Add(team);

await \_context.SaveChangesAsync();

var addedCapToTeam = new UserInTeam { TeamId = team.Id, UserId = team.CaptainId };

\_context.UsersInTeams.Add(addedCapToTeam);

await \_context.SaveChangesAsync();

return Ok(new { message = "Команда создана", team});

}

[HttpPost("{teamId}/add-user")]

public async Task<IActionResult> AddUserToTeam(int teamId, [FromBody] UserToTeamDto dto)

{

var team = await \_context.Teams.FirstOrDefaultAsync(t => t.Id == teamId);

if (team == null)

return NotFound("Команда не найдена.");

var user = await \_context.Users.FindAsync(dto.UserId);

if (user == null)

return NotFound("Пользователь не найден.");

var userInTeam = await \_context.UsersInTeams.FirstOrDefaultAsync(t => t.UserId == dto.UserId);

if (userInTeam != null)

return BadRequest("Пользователь состоит в команде.");

var addedUser = new UserInTeam { TeamId = team.Id, UserId = user.Id };

\_context.UsersInTeams.Add(addedUser);

await \_context.SaveChangesAsync();

return Ok(new { message = "Пользователь добавлен в команду", team.Name });

}

[HttpDelete("remove-user/{userId}")]

public async Task<IActionResult> RemoveUserFromTeam(int userId)

{

var user = await \_context.UsersInTeams.FirstOrDefaultAsync(u => u.UserId == userId);

if (user == null)

return BadRequest("Пользователь не состоит в команде.");

\_context.UsersInTeams.Remove(user);

await \_context.SaveChangesAsync();

return Ok("Пользователь был успешно удалён из вашей команды.");

}

[HttpDelete("remove/{teamId}")]

public async Task<IActionResult> RemoveTeam(int teamId)

{

var team = await \_context.Teams.FindAsync(teamId);

if (team == null)

return NotFound("Такой команды не существует.");

var userInTeam = await \_context.UsersInTeams.FirstOrDefaultAsync(t => t.TeamId == teamId);

if (userInTeam != null)

return BadRequest("В команде состоят пользователи.");

\_context.Teams.Remove(team);

await \_context.SaveChangesAsync();

return Ok("Команда успешно удалена.");

}

}

}

**TournamentController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Net.WebSockets;

using Turaev.WebAPI.Data;

using Turaev.WebAPI.Models;

namespace Turaev.WebAPI.Controllers

{

[Route("api/tournament")]

[ApiController]

public class TournamentController : ControllerBase

{

private readonly TuraevDbContext \_context;

public TournamentController(TuraevDbContext context)

{

\_context = context;

}

[HttpPost]

public async Task<IActionResult> CreateTournament([FromBody] TournamentCreateDto dto)

{

var tournament = new Tournament

{

Name = dto.Name,

Description = dto.Description,

StartDate = dto.StartDate

};

\_context.Tournaments.Add(tournament);

await \_context.SaveChangesAsync();

return Ok(new { message = "Турнир успешно создан" });

}

[HttpGet]

public async Task<IActionResult> GetAllTournament()

{

var tournaments = await \_context.Tournaments.ToListAsync();

return Ok(tournaments);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetTournamentById(int id)

{

var tournament = await \_context.Tournaments.FirstOrDefaultAsync(t => t.Id == id);

if (tournament == null)

return NotFound("Турнир не найден.");

return Ok(tournament);

}

[HttpGet("{tournamentId}/members")]

public ActionResult<IEnumerable<TeamInTournament>> GetTournamentMembersById(int tournamentId)

{

var members = \_context.TeamsInTournamnets.Where(t => t.TournamentId == tournamentId).ToList();

return Ok(members);

}

[HttpPost("{tournamentId}/add-team/{teamId}")]

public async Task<IActionResult> AddTeamToTournament(int tournamentId, int teamId)

{

var tournament = await \_context.Tournaments.FirstOrDefaultAsync(t => t.Id == tournamentId);

if (tournament == null)

return NotFound("Турнир не найден.");

var team = await \_context.Teams.FindAsync(teamId);

if (team == null)

return NotFound("Команда не найдена.");

var teamInTournament = await \_context.TeamsInTournamnets.FirstOrDefaultAsync(t => t.TournamentId == tournamentId && t.TeamId == teamId);

if (teamInTournament != null)

{

if (teamInTournament.TournamentId == tournamentId && teamInTournament.TeamId == teamId)

return BadRequest("Команда уже участвует в турнире.");

}

var dto = new TeamInTournament

{

TeamId = teamId,

TournamentId = tournamentId

};

\_context.TeamsInTournamnets.Add(dto);

await \_context.SaveChangesAsync();

return Ok(new {message = "Команда успешно добавлена в турнир", tournament});

}

[HttpDelete("{tournamentId}/remove-team/{teamId}")]

public async Task<IActionResult> RemoveTeamToTournament(int tournamentId, int teamId)

{

var tournament = await \_context.Tournaments.FirstOrDefaultAsync(t => t.Id == tournamentId);

if (tournament == null)

return NotFound("Турнир не найден.");

var team = await \_context.Teams.FindAsync(teamId);

if (team == null)

return NotFound("Команда не найдена.");

var teamInTournament = await \_context.TeamsInTournamnets.FirstOrDefaultAsync(t => t.TournamentId == tournamentId && t.TeamId == teamId);

if (teamInTournament == null)

{

return BadRequest("Команда не состоит в этом турнире.");

}

var dto = await \_context.TeamsInTournamnets.FindAsync(teamInTournament.Id);

\_context.TeamsInTournamnets.Remove(dto);

await \_context.SaveChangesAsync();

return Ok(new {message = "Команда успешно исключена из турнира."});

}

[HttpPost("{tournamentId}/generate-bracket")]

public async Task<IActionResult> GenerateBracket(int tournamentId)

{

var teams = \_context.TeamsInTournamnets

.Where(t => t.TournamentId == tournamentId)

.Select(t => t.TeamId)

.ToList();

if (teams.Count < 2)

{

return BadRequest("Недостаточно команд для создания турнира.");

}

var matches = new List<Match>();

var brackets = new List<Bracket>();

int round = 1;

while (teams.Count > 1)

{

var nextRoundTeams = new List<int>();

for (int i = 0; i < teams.Count; i += 2)

{

int teamOneId = teams[i];

int? teamTwoId = (i + 1 < teams.Count) ? teams[i + 1] : null;

var match = new Match

{

TournamentId = tournamentId,

MatchDate = DateTime.UtcNow.AddDays(round),

Status = "Scheduled",

ScoreTeamOne = 0,

ScoreTeamTwo = 0

};

\_context.Matches.Add(match);

await \_context.SaveChangesAsync();

\_context.TeamsInMatches.Add(new TeamInMatch

{

MatchId = match.Id,

TeamOneId = teamOneId,

TeamTwoId = teamTwoId

});

await \_context.SaveChangesAsync();

var bracket = new Bracket

{

TournamentId = tournamentId,

Round = round,

MatchId = match.Id,

WinnerRoundId = null

};

\_context.Brackets.Add(bracket);

await \_context.SaveChangesAsync();

}

break;

}

await \_context.SaveChangesAsync();

return Ok("Турнирная сетка успешно создана.");

}

[HttpPost("{matchId}/set-result")]

public async Task<IActionResult> SetMatchResult(int matchId, [FromBody] MatchResultDto result)

{

var match = await \_context.Matches.FindAsync(matchId);

if (match == null)

{

return NotFound("Матч не найден.");

}

match.ScoreTeamOne = result.ScoreTeamOne;

match.ScoreTeamTwo = result.ScoreTeamTwo;

match.Status = "Completed";

await \_context.SaveChangesAsync();

int winnerId = result.ScoreTeamOne > result.ScoreTeamTwo ? result.TeamOneId : result.TeamTwoId;

var bracket = await \_context.Brackets.FirstOrDefaultAsync(b => b.MatchId == matchId);

if (bracket != null)

{

bracket.WinnerRoundId = winnerId;

await \_context.SaveChangesAsync();

}

var currentRoundBrackets = await \_context.Brackets

.Where(b => b.TournamentId == match.TournamentId && b.Round == bracket.Round)

.ToListAsync();

if (currentRoundBrackets.All(b => b.WinnerRoundId != null))

{

int nextRound = bracket.Round + 1;

var winners = currentRoundBrackets.Select(b => b.WinnerRoundId.Value).ToList();

if (winners.Count == 1)

{

var tournament = await \_context.Tournaments.FindAsync(match.TournamentId);

if (tournament != null)

{

tournament.WinnerId = winners.First();

tournament.EndDate = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

return Ok("Турнир завершен. Победитель: " + winners.First());

}

for (int i = 0; i < winners.Count; i += 2)

{

int teamOneId = winners[i];

int? teamTwoId = (i + 1 < winners.Count) ? winners[i + 1] : null;

var newMatch = new Match

{

TournamentId = match.TournamentId,

MatchDate = DateTime.UtcNow.AddDays(nextRound),

Status = "Scheduled",

ScoreTeamOne = 0,

ScoreTeamTwo = 0

};

\_context.Matches.Add(newMatch);

await \_context.SaveChangesAsync();

\_context.TeamsInMatches.Add(new TeamInMatch

{

MatchId = newMatch.Id,

TeamOneId = teamOneId,

TeamTwoId = teamTwoId

});

await \_context.SaveChangesAsync();

var newBracket = new Bracket

{

TournamentId = match.TournamentId,

Round = nextRound,

MatchId = newMatch.Id,

WinnerRoundId = null

};

\_context.Brackets.Add(newBracket);

await \_context.SaveChangesAsync();

}

}

return Ok("Результат матча сохранен и победитель продвинут.");

}

}

}

**UserController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Turaev.WebAPI.Data;

using Turaev.WebAPI.Models;

namespace Turaev.WebAPI.Controllers

{

[Route("api/user")]

[ApiController]

public class UserController : ControllerBase

{

private readonly TuraevDbContext \_context;

public UserController(TuraevDbContext context)

{

\_context = context;

}

[HttpGet("all")]

public ActionResult<IEnumerable<User>> GetAllUsers()

{

return Ok(\_context.Users.ToList());

}

[HttpPut("update/{id}")]

public async Task<IActionResult> UpdateUser(int id, [FromBody] UserUpdateDto request)

{

if (request == null)

return BadRequest("Данные не предоставлены.");

var user = await \_context.Users.FindAsync(id);

if (user == null)

return NotFound("Пользователь не найден.");

user.Name = request.Name ?? user.Name;

user.Email = request.Email ?? user.Email;

\_context.Users.Update(user);

await \_context.SaveChangesAsync();

return Ok(new {message = "Данные пользователя обновлены успешно."});

}

}

}