

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»

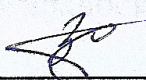
Профиль подготовки: «Программное обеспечение средств
вычислительной техники и автоматизированных
систем»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

БАКАЛАВРА

на тему

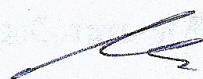
**МИКРОСЕРВИС УЧЕТА СПОРТИВНЫХ
КОМАНД ПРИ ПРОВЕДЕНИИ ТУРНИРОВ**

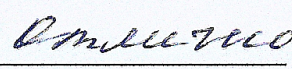
Студент  31.05.2022 г. Здерев С.Г.
(подпись, дата)

Руководитель  31.05.2022 г. Синев М.П.
(подпись, дата)

Нормоконтролёр  31.05.2022 г. Бычков А.С.
(подпись, дата)

Работа допущена к защите (протокол зас. кафедры от 02.06.2022 г. № 13)

Заведующий кафедрой  М.А. Митрохин
(подпись)

Работа защищена с отметкой  (протокол заседания ГЭК от
15.06.2022 № 8)

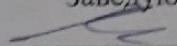
Секретарь ГЭК  Е.И. Гурин
(подпись)

Пенза 2022

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

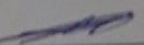
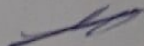
Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
 М.А. Митрохин
«26» апреля 2022 г.

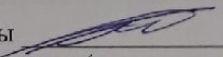
**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВАРА**

1. Студент Здерев Сергей Геннадьевич, гр.18BB2 факультета вычислительной техники.
2. Тема работы: «Микросервис учета спортивных команд при проведении турниров»
Тема утверждена приказом ПГУ №357/о от "26" апреля 2022 г.
3. Руководитель бакалаврской работы к.т.н., доцент каф. ВТ ПГУ Синев М.П.
4. Задание на работу (назначение разработки, исходные данные и т.п.)
Микросервис предназначен для ведения статистики по спортивным мероприятиям, учитывающим команды по различным видам спорта.
5. Перечень подлежащих разработке вопросов
Требуется произвести анализ предметной области, выбрать среду программирования, базы данных и язык программирования для приложения. Изучить особенностей разработки под web-приложения, разработать *REST API*. Также выполнить задание по БЖД.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Анализ предметной области	20	26.04.2022 – 29.04.2022	
2.	Выбор среды, базы данных и языка программирования	20	30.04.2022 – 02.05.2022	
3.	Изучение особенности разработки под web приложения	51	03.05.2022 – 07.05.2022	
4.	Разработка базы данных	30	08.05.2022 – 15.05.2022	
5.	Разработка REST API	46	16.05.2022 – 19.05.2022	
6.	Разработка программной части	111	20.05.2022 – 27.05.2022	
7.	Оформление пояснительной записки	46	28.05.2022 - 31.05.2022	

Дата выдачи задания "26" апреля 2022 г.

Руководитель бакалаврской работы  26.04.2022 Синева М.П.
(подпись, дата)

Задание к исполнению принял студент  26.04.2022 Здерев С.Г.
(подпись, дата)

Работу к защите допустить
Декан факультета  Фионова Л.Р.
10.06.2022

Реферат

Работа содержит 101 лист, 19 рисунков, 6 таблиц, 9 источников, 5 приложений.

МИКРОСЕРВИС УЧЁТА СПОРТИВНЫХ КОМАНД ПРИ ПРОВЕДЕНИИ ТУРНИРОВ

Цель ВКР – разработать *web*-приложение, позволяющее пользователям получать информацию о спортивных командах.

Разработанный микросервис предназначен для ведения статистики по спортивным мероприятиям, учитывающим команды по различным видам спорта.

В процессе выполнения работы производилось:

- изучение особенностей разработки под *web*-приложения;
- разработка базы данных;
- разработка REST API;
- разработка программной части.

В результате выполнения работы было создано приложение, удовлетворяющее поставленной задаче.

					ПГУ 1.09.03.01.07.001 ПЗ		
Изм.	ЛистЛ	№ докум.№	Подпись	Дата			
Разраб.		Здерева С.Г.			Лит.	Лист	Листов
Провер.		Синев М.П.				2	84
Н. Контр.		Бычков А.С.			ФВТ, каф. ВТ		
Утверд.		Митрохин М.А.			гр. 18ВВ2		
					Микросервис учёта спортивных команд		
					Пояснительная записка		

Содержание

Перечень принятых сокращений и обозначений	4
Введение.....	5
1 Микросервис учета спортивных команд при проведении турниров.....	6
1.1 Анализ аналогов, постановка задачи	6
1.1.1 Анализ аналогов	6
1.1.2 Постановка задачи.....	8
1.2 Анализ предметной области	9
1.2.1 Архитектура.....	9
1.2.1.1 Описание архитектуры <i>REST API</i>	9
1.2.2.2 Выбор реляционной СУБД	14
1.2.2.3 Выбор языка программирования	18
1.2.2.4 Анализ фреймворков	19
1.2.2.5 Описание <i>Swagger</i>	22
1.3 Реализация.....	26
1.3.1 Структура БД.....	26
1.3.2 Описание <i>REST API</i>	27
1.3.3 Описание классов	29
1.3.4 Программная реализация.....	30
1.4 Описание результата.....	31
2 Безопасность жизнедеятельности.....	37
2.1 Расчет эквивалентного уровня звука на рабочих местах в компьютерных залах, офисах и других помещениях, где установлены ПЭВМ	37
Заключение	44
Приложение А – UML диаграммы	46
Приложение Б – Модель базы данных.....	50
Приложение В – Описание REST API	52
Приложение Г – Листинг программы	58
Приложение Д – Презентация.....	94

Перечень принятых сокращений и обозначений

API (Application Programming Interface) - интерфейс прикладных программ

HTML (HyperText Markup Language) - язык разметки гипертекста

HTTP (HyperText Transfer Protocol) - протокол передачи гипертекста

ID (Identifier) - идентификатор

JSON (JavaScript Object Notation) - формат обмена текстовыми данными на основе *JavaScript*

NoSQL (не только *SQL*) - не просто *SQL*

OLAP (Online Analytical Processing) - интерактивная аналитическая обработка

REST (Representational State Transfer) - передача репрезентативных состояний

SQL (Structured Query Language) - структурированный язык запросов

UML (Unified Modelling Language) - унифицированный язык моделирования

URI (Uniform Resource Identifier) - однозначная идентификация ресурсов

XML (eXtensible Markup Language) - расширяемый язык разметки

БД - база данных

ОРСУБД - объектно-реляционная система управления базами данных

РСУБД - реляционная система управления базами данных

СУБД - система управления базами данных

Введение

В наши дни *web*-приложения являются неотъемлемой частью жизни современного человека. Большинство крупных компаний активно используют сайты для продвижения своей продукции. Для стабильной работы многие разработчики используют различные архитектуры разработки приложений [1, 2].

Одной из наиболее распространенных и комфортных архитектур является микросервисная архитектура. Это выгодно, поскольку приложение разделено на функциональные части - микросервисы, - каждая из которых может быть расширена для распределения нагрузки и повышения стабильности всего приложения.

Спорт является неотъемлемой частью жизни большинства людей, и многие болельщики с нетерпением ждут результатов матчей команд в различных видах спорта. Это проблема, как для профессионалов, так и для любителей. Не секрет, что дети и молодежь уделяют большое внимание спорту. Это приложение было разработано специально для этой цели, поскольку многим тренерам было бы сложно организовать соревнования на школьном и университетском уровне без автоматизированной статистической системы.

В данной работе описывается реализация *CRUD*-приложения, реализующего *RESTful* архитектуру для распределенного, масштабируемого веб-сервиса, который может служить основой для будущих расширений. Функциональная направленность сервиса заключается в хранении и обработке информации о спортивных лигах, командах и игроках.

1 Микросервис учета спортивных команд при проведении турниров

1.1 Анализ аналогов, постановка задачи

По мере развития технологий и Интернета все больше информации хранится онлайн в самых разных местах. Спортивные сайты не являются исключением, и все больше болельщиков используют их вместо газет и телевидения, чтобы узнать больше о своих любимых командах и спортсменах.

Приложение осуществляет обработку информации о спорте.

При разработке программного продукта важно точно нацелиться на рынок, проанализировать его и выявить положительные и отрицательные стороны различных конкурентов. Приложение позволяет осуществить *REST*-взаимодействие с остальными сервисами и в результате данного взаимодействия осуществляет изменения в базе данных. В данной работе описывается бэкенд микросервиса, отвечающего за сбор статистики по видам спорта, командам и игрокам.

1.1.1 Анализ аналогов

Предлагаю рассмотреть следующие сайты с точки зрения рядового пользователя:

MyScore.ru - это сервис, который предлагает в основном онлайн результаты матчей и статистическую информацию о командах и игроках, рисунок 1. Сайт предоставляет всю основную статистику, что делает его очень популярным среди многих любителей спорта, и обновления счета в режиме реального времени.

На *MyScore.com* представлены представители практически всех без исключения видов спорта и лиг. Вы можете получить статистику, выбрав матч из списка матчей, сыгранных сегодня или в ближайшие несколько дней. Вы также можете просмотреть конкретную лигу, чемпионат или турнир в списке слева.

Преимущества:

- объёмность;

					ПГУ1 090301.04.001 ПЗ	Лист
						6
Изм.	Лист	№ докум.	Подпись	Дата		

- понятный, минималистичный интерфейс.

Минусы:

- отсутствие разбора спортивных команд в каждом виде спорта;
- скудные возможности фильтрации.

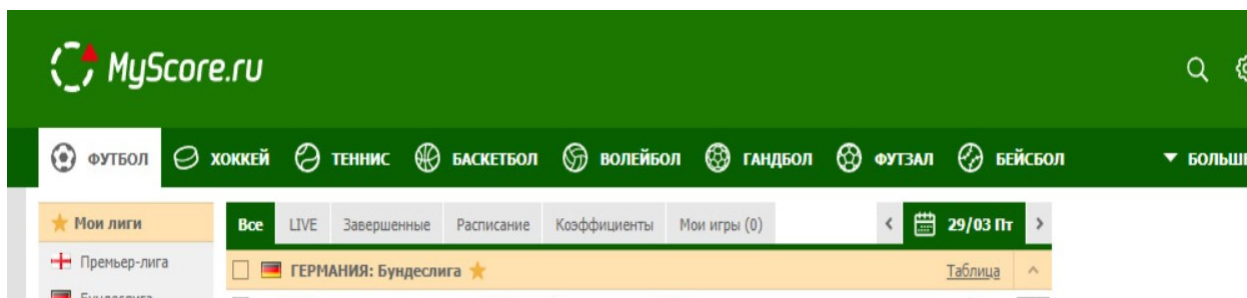


Рисунок 1 - Приложение «MyScore.ru»

Информационный онлайн ресурс "R-Hockey.ru", рисунок 2, включает в себя спортивную дисциплину - хоккей, действующие лиги и т.д. Функции выбора спортивных лиг, спортивных команд и игроков с широкой линией статистики.

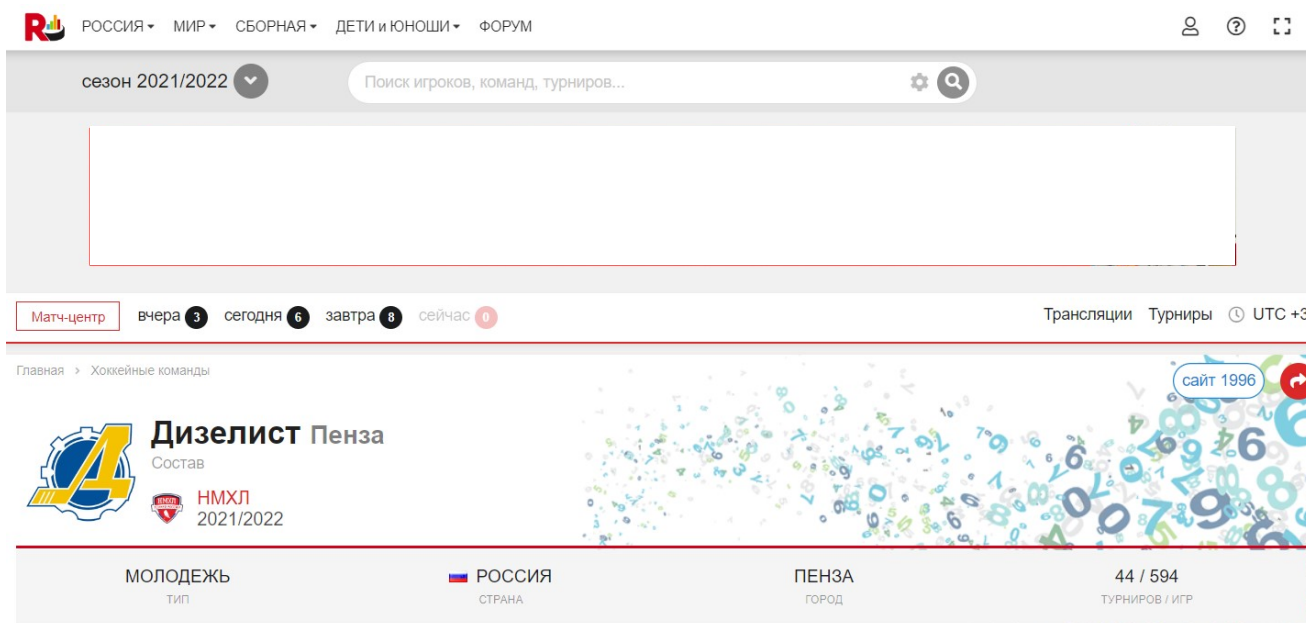


Рисунок 2 - Приложение «R-Hockey.ru»

Плюсы:

- широкая статистика по каждому игроку с разделением на роли;
- разновидность лиг, начиная с юношеских.

Минусы:

- отсутствие разбора спортивных команд в каждом виде спорта;
- наполненность ресурса.

Портал "ВХЛ" – высшая хоккейная лига, рисунок 3. С алфавитным анализом всех видов спорта, поиском по названию и всем лигам.

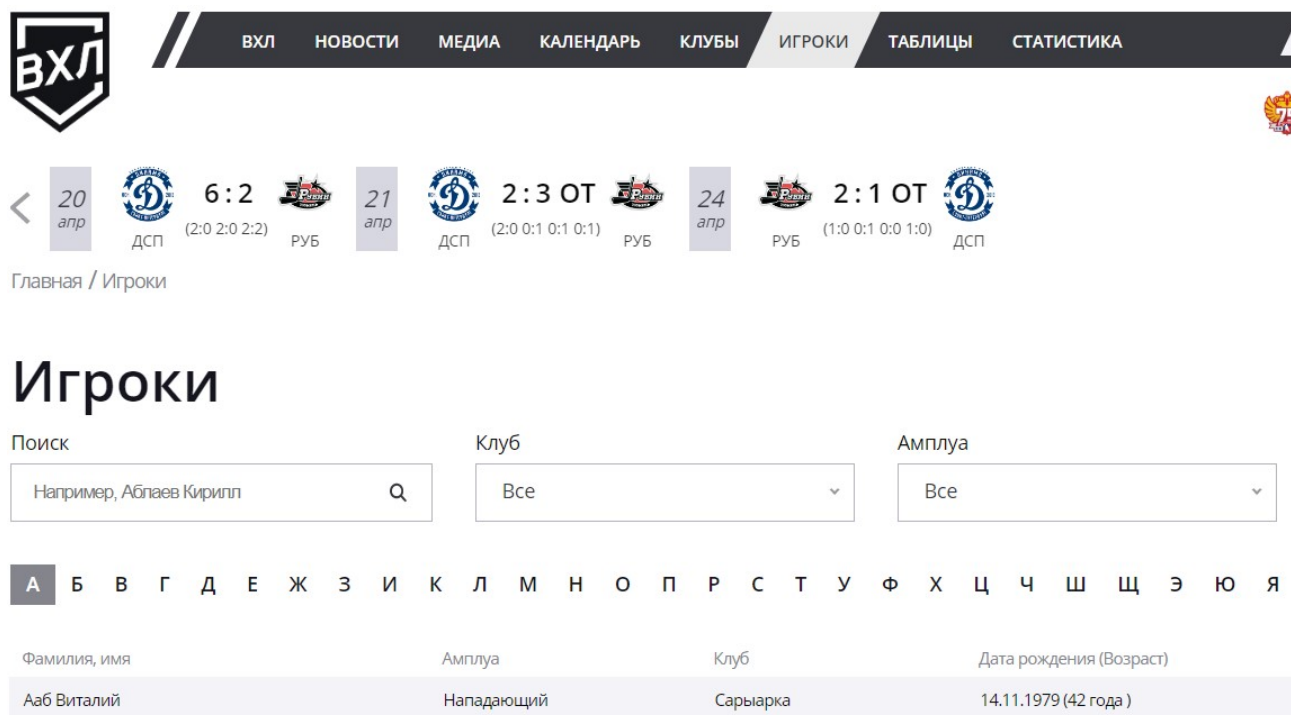


Рисунок 3 - Приложение «ВХЛ»

Плюсы:

- обширный спектр фильтрации списка игроков по всем параметрам и алфавиту;
- широкий функционал поиска.

Минусы:

- отсутствие разбора спортивных команд в каждом виде спорта;
- примитивный интерфейс.

1.1.2 Постановка задачи

Исходя из рассмотренных сайтов, можно сделать следующие выводы: приложение должно быть максимально понятным и простым для восприятия, возможность фильтрации данных по различным параметрам, понятный поиск и

полнота информации с возможностью отметки мероприятий. Приложение будет реализовывать *beck-end* часть взаимодействия клиент-сервера, развёртку базы данных и реализацию *REST full* архитектуры.

В основе проекта будет лежать две роли, пользователь и администратор, изображены в приложении А.1. Первый и самый основной будет проверять статистику спортивных встреч, анализировать, оставлять комментарии к тем. Администратор - как пользователь с более широким спектром доступа. Он будет иметь возможность добавлять, информацию и корректировать поступающий контент на сайт.

1.2 Анализ предметной области

При разработке программного продукта важно рассмотреть варианты решения проблем. В этой главе обсуждаются общие теоретические вопросы, связанные с техническими элементами, необходимыми для внедрения корпоративного программного обеспечения. К ним относятся выбор системы баз данных, архитектура программного обеспечения, языки программирования и архитектура *API*. В каждом подразделе обосновывается выбор того или иного технического решения проблемы.

1.2.1 Архитектура

1.2.1.1 Описание архитектуры *REST API*

REST - это архитектурный стиль, используемый для связи между компонентами распределенных веб-приложений. При разработке *REST*-приложений обязательными являются шесть ограничений.

Модель клиент-сервер. Система должна представлять собой автономный клиент-сервер, т.е. это ограничение основано на принципе параллельных приложений. Это упрощает серверную часть приложения и облегчает будущие расширения. Клиент-серверный интерфейс также позволяет переносить его на другие платформы. Сервер представляет собой микросервис для регистрации спортивных команд.

Отсутствие состояний. Взаимодействие между клиентом и сервером

					ПГУ1 090301.04.001 ПЗ	Лист
						9
Изм.	Лист	№ докум.	Подпись	Дата		

основано на следующих условиях: данные клиента не должны храниться на сервере между запросами, а каждый запрос клиента должен отправлять все соответствующие данные транзакции на сервер перед выполнением запроса. Клиент также должен хранить информацию о состоянии сессии. Ожидается, что запросы клиентов будут находиться во временном состоянии во время обработки. Это ограничение освобождает ресурсы сервера, так как информацию о состоянии не нужно хранить, и улучшает масштабируемость системы, так как несколько клиентов могут обрабатываться одновременно. Средства учета микросервисов не регистрируют клиентские сессии и поэтому не знают об их состоянии.

Сохранение памяти. Клиенты могут кэшировать ответы сервера. В этом случае ответ сервера следует кэшировать, если получены новые данные, а не если данные уже были получены. Хорошо спроектированный кэш может улучшить производительность и масштабируемость всей распределенной веб-системы, ограничивая взаимодействие клиента и сервера.

Многослойные системы. Архитектура *REST* позволяет разделить систему на уровни, предполагая, что каждый элемент системы ссылается только на элементы следующего уровня. Выделенные агенты могут быть использованы для улучшения масштабируемости за счет балансировки нагрузки и совместного использования кэша. Политики безопасности также могут быть использованы для обеспечения целостности.

Код по требованию. Это необязательное требование, архитектура *REST* позволяет загружать программное обеспечение (код) с клиента.

1.2.1.2 Выбор архитектуры приложения

Существует два основных типа архитектуры для проектирования веб-приложений:

- монолит.
- микросервис.

Чтобы понять, какую архитектуру выбрать, необходимо проанализировать характеристики различных архитектур.

Монолитная архитектура приложений.

Монолитная архитектура - это подход к разработке приложений, при котором приложения проектируются как большие взаимосвязанные модули. Модульные компоненты связаны между собой и совместно используют память и ресурсы. Рисунок 4 наглядно иллюстрирует монолитную архитектуру.

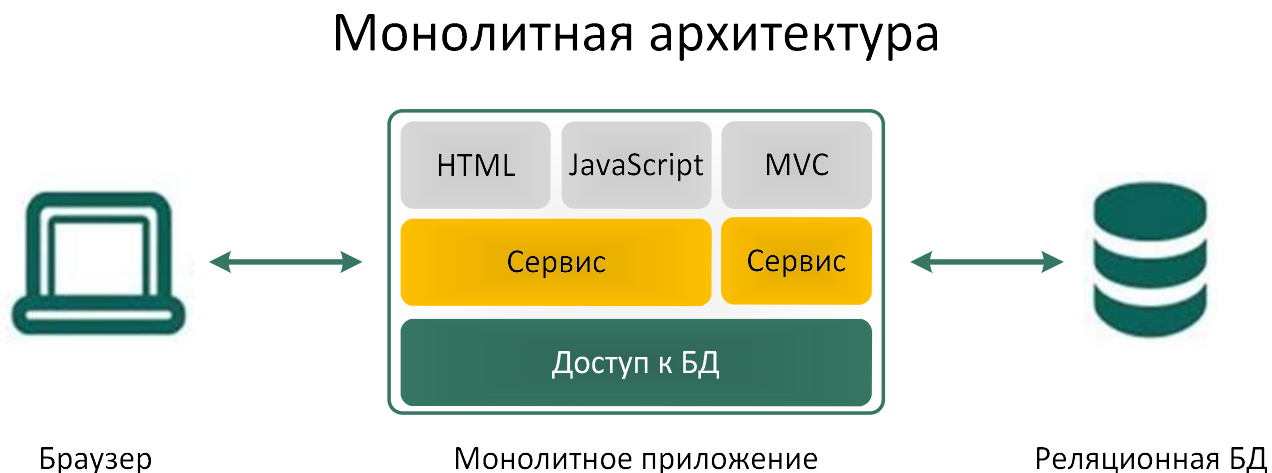


Рисунок 4 - Монолитная архитектура

Преимущества монолитной архитектуры заключаются в следующем:

- его относительно легко внедрить, установить и управлять им.
- согласованность кода легче поддерживать и отлаживать.
- облегчает добавление общей функциональности к компонентам.
- он обеспечивает достаточную производительность для малых и средних приложений.

Хотя монолитные архитектуры имеют очевидные преимущества, у них есть и существенные недостатки:

- повышенная сложность и операционная неопределенность.
- по мере роста приложений становится все сложнее изолировать сервисы, расширять каждый сервис отдельно и поддерживать код.
- изменения сложны и требуют много времени.
- если одна служба выходит из строя, может выйти из строя все приложение.

Второй недостаток заключается в том, что отказ одного сервиса приводит к отказу всего приложения, что неприемлемо с точки зрения бизнеса.

Архитектура приложений для микросервисов.

Микросервисная архитектура является современным выражением сервис-ориентированной архитектуры и используется при проектировании распределенных систем [3]. Микросервисные приложения должны состоять из небольших независимых компонентов (микросервисов), каждый из которых может независимо развиваться и масштабироваться, и взаимодействовать со своими собственными ресурсами. Отдельные микросервисы могут располагаться на разных серверных узлах. Схематичное сравнение микросервисной и монолитной архитектуры представлено на рисунке 5.

Преимущества использования микросервисов:

- возможность обновления приложения по частям, отдельная установка;
- если микросервис выходит из строя, это не затрагивает все приложение;
- если произойдет сбой, приложение не будет повреждено и вряд ли будет потеряно;
- архитектура микросервисов обеспечивает логическое разделение и позволяет размещать различные модули на разных серверах;
- возможность использования нескольких языков программирования путем установки модулей на разных серверах;
- различные методы выбора оптимальной функциональности;
- распределенное управление;
- распределенное управление данными.

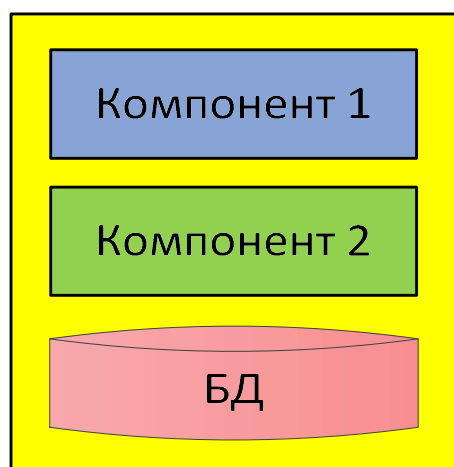
Как и все технологии и методы разработки, микросервисы имеют некоторые недостатки.

- сложность реализации;
- поскольку они используют *API* и развернуты на разных серверных узлах, ими необходимо тщательно управлять;
- необходимость четкой дифференциации между модулями;
- увеличение задержки в сети.

Заключение: несмотря на сложность внедрения и управления, архитектура

микросервисов подходит для реализации функциональности микросервисов в бухгалтерии спортивной команды.

Монолит



Микросервисы

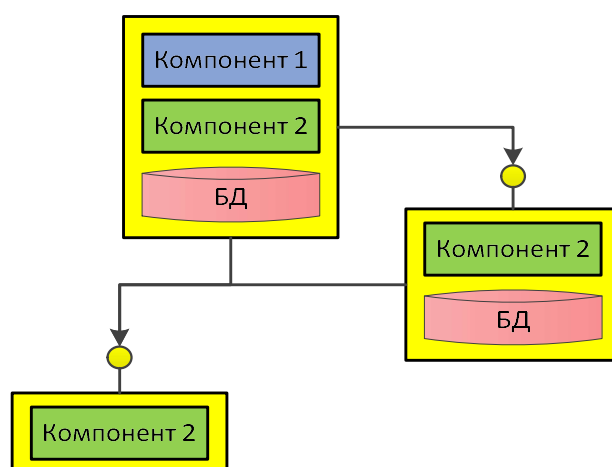


Рисунок 5 - Сравнение монолитной и микросервисной архитектур

1.2.2 Выбор инструментов разработки

1.2.2.1 Анализ типов СУБД

Существует два типа систем баз данных: реляционные (*SQL*) и нереляционные (*NoSQL*). Для того чтобы сделать выбор между различными типами баз данных, необходимо знать их различия.

Существует шесть различий, каждое из которых будет рассмотрено более подробно.

Различия касаются структуры и типа хранимых данных. Реляционные базы данных должны иметь четко определенную структуру для хранения данных, в то время как базы данных *NoSQL* не имеют таких строгих ограничений и могут хранить данные любого типа. Структура данных с четко определенными отношениями между объектами таблицы подходит для микросервисов спортивных команд во время игр.

Разница в структуре запросов: к базам данных *SQL* можно обращаться только с помощью *SQL*, реляционные системы управления базами данных опираются на стандарты *SQL*. В нереляционных базах данных нет четкого

определения того, как должен выглядеть запрос, поэтому каждая база данных *NoSQL* использует свой собственный способ работы с данными. Микросервис "Расчет спортивных команд" требует создания запроса с несколькими условными вариантами выбора, и использование нереляционной базы данных вызывает проблемы при создании запроса, поэтому реляционная база данных должна быть выбрана явно.

Масштабируемость. Оба типа баз данных вертикально масштабируются, но нереляционную базу данных лучше использовать горизонтально. Для микросервиса, который предоставляет отчеты о спортивных командах во время турниров, имеет смысл выбрать решение с лучшей горизонтальной масштабируемостью.

Надежность. Базы данных *SQL* более надежны, чем базы данных *NoSQL*, поэтому реляционные базы данных лучше подходят для микросервисных решений.

Поддержка: реляционные системы баз данных существовали задолго до систем *NoSQL*. В настоящее время доступны как бесплатные, так и платные решения, готовые и поддерживаемые. Выбор снова сделан в пользу *SQL*.

Базы данных *SQL* являются более мощными, чем нереляционные системы баз данных, поскольку они изначально были разработаны для точной обработки сложных структур данных. Базы данных *SQL* также лучше подходят для микроуправления счетами спортивных команд.

В заключение анализа характеристик реляционных и нереляционных систем баз данных, для решения проблемы микроменеджмента спортивной команды была выбрана реляционная система баз данных.

1.2.2.2 Выбор реляционной СУБД

Чтобы выбрать наиболее подходящую систему баз данных, необходимо проанализировать популярные и появляющиеся на рынке базы данных *SQL*. Это *MySQL*, *PostgreSQL*, *MS SQL Server* и *Oracle database*.

MS SQL Server.

Microsoft SQL Server - это система управления реляционными базами

данных, разработанная корпорацией Майкрософт. Основным языком для написания запросов является язык *Transact-SQL*. Эта система баз данных подходит как для создания персональных баз данных, так и для внедрения баз данных на уровне предприятия. Последней версией продукта является *SQL Server 2016*. В этой версии вы можете отслеживать эволюцию данных с течением времени (поддержка переменных данных). Эта версия также поддерживает динамическую маскировку данных, которая гарантирует, что только авторизованные пользователи могут получить доступ к конфиденциальным данным.

Преимущества *MS SQL Server*:

- простота в использовании;
- возможность настройки и управления уровнями производительности для снижения рабочей нагрузки;
- возможность настройки и управления функциями *SQL Server*;
- возможность интеграции с другими продуктами *Microsoft*;
- возможность просмотра на мобильных устройствах.

Недостатки СУБД *MS SQL Server*:

- очень высокая стоимость продукта для юридических лиц;
- возможные проблемы со службами интеграции импорта файлов;
- высокое потребление ресурсов *SQL server* (даже если настроена производительность).

В дополнение ко всем этим преимуществам, эта система баз данных имеет один очень важный недостаток, который делает ее использование невозможным: высокие затраты для юридических лиц. В бесплатной версии недостаточно функций для реализации базы данных микросервисов.

Oracle database - это золотая система управления базами данных Oracle. История этой СУБД восходит к концу 70-х годов 20-ого века. Последней версией продукта, выпущенной в 2021 году, является версия 21С ("С" означает "облако"). Эта версия поддерживает подключение к базе данных через облачные сервисы, что позволяет размещать базу данных на нескольких серверах и хранить большие

объемы данных. наличие интегрированных баз данных позволяет выполнять многопользовательские операции ("арендаторы", обслуживающие несколько клиентских организаций) и переносить базы данных в режиме реального времени. Повышенная безопасность транзакций - каждая транзакция изолирована от других.

Преимущества *Oracle Database*:

- надежность;
- расширенные функции и инновации в последних версиях.

Недостатки базы данных *Oracle*:

- высокая стоимость продукции для юридических лиц;
- система требует аппаратных ресурсов с достаточной емкостью и производительностью.

К сожалению, высокая цена не позволяет использовать эту ОРСУБД. Бесплатная версия позволяет использовать небольшие базы данных.

MySQL - это бесплатная система управления реляционными базами данных. *Oracle* приобрела права на этот бренд в 2010 году. Поскольку *MySQL* является свободным программным обеспечением, регулярно выпускаются новые версии и добавляются новые функции. Для коммерческих организаций также доступны платные версии. Бесплатная версия ориентирована на надежность и скорость, но не на функциональность. *MySQL* - это очень гибкая система баз данных, поскольку она может работать с различными типами таблиц.

Преимущества *MySQL*:

- для личного пользования бесплатно;
- имеет хорошую документацию;
- безопасность;
- возможность использования множества функций в бесплатной версии;
- совместимость с другими базами данных, например *DB2* и *Oracle*.

Недостатки *MySQL*:

- проблемы с надежностью;
- платная для коммерческого использования.

- некоторые функциональные ограничения;
- отсутствует поддержка *XML* и *OLAP*;
- ограниченная поддержка бесплатной версии, доступна только платная поддержка.

Несмотря на обнаруженные преимущества, система в основном используется для небольших и средних проектов, она не подходит для крупных проектов и не является бесплатной.

PostgreSQL - это бесплатная система управления реляционными базами данных с открытым исходным кодом, разработанная сообществом *PostgreSQL* [4]. Он основан на языке *SQL*. Последняя доступная версия - 14.2. *PostgreSQL* активно развивается сообществом, и постоянно добавляются новые функции. В последнее время он все чаще используется в крупных коммерческих проектах.

PostgreSQL обладает следующими преимуществами:

- бесплатная и для личного и для коммерческого использования;
- хорошо масштабируется;
- возможность обрабатывать терабайты данных;
- поддержка формата *JSON*;
- существует множество готовых, встроенных функций;
- хорошая документация;
- большое сообщество с поддержкой.

Недостатки *PostgreSQL*:

- неэффективная репликация данных;
- трудности с обновлением.

PostgreSQL выигрывает по всем параметрам и самое главное – программа бесплатно распространяется. Поэтому было принято решение использовать именно эту ОРСУБД.

В проекте используется версия 14.2, на момент начала разработки она была выбрана как наиболее стабильная.

В приложении Б изображена физическая модель базы данных, в её состав входит четыре сущности, которые объединены между собой связями

ManyToOne и *OneToMany*, соответственно игрок либо команда не может существовать без определённого вида спорта и т.д.

1.2.2.3 Выбор языка программирования

При реализации любого программного продукта одним из самых важных этапов является выбор языка программирования. От этого зависит продуктивность и чистота кода, а также длительность периода создания минимально жизнеспособного продукта.

В настоящее время наиболее популярными языками программирования для разработки веб-приложений являются: *Go*, *Java*, *JavaScript*, *Python* [5]. Такие языки как *C++*, *Php* и *Ruby* в последнее время теряют популярность в веб-разработке.

Язык программирования *Go*.

Go (также известный как *Golang*) — компилируемый многопоточный язык программирования, разработанный компанией *Google*. Язык разрабатывался для проектирования высокопроизводительных программ и для решения проблем большинства языков. Язык проектировался с целью нивелировать такие проблемы при разработке как медленная сборка программы, сложность при разработке инструментария, проблемы межъязыкового взаимодействия и другие.

Но данный язык не является объекто-ориентированным в полном смысле этого понятия, так как в язык сознательно не была включена концепция наследования. Язык обладает меньшей переносимостью на разные платформы, так как компилируется в машинный код.

Язык программирования *Python*.

Python – высокоуровневый язык программирования, который ориентирован на повышение производительности разработчика и читаемости кода. Этот язык очень гибкий и обладает широким функционалом, а код высокой степенью читаемости. Но у данного языка есть существенный недостаток – он обладает низким быстродействием и поэтому часто используется как язык второго плана.

Язык программирования *JavaScript*.

JavaScript – мультипарадигменный язык программирования. Язык не

принадлежит какой-либо организации или компании. Наиболее часто используется в качестве языка сценариев *web*-страниц. *JavaScript* поддерживает императивный, декларативный и объектно-ориентированный стили программирования. *JavaScript* часто используется в таких программных продуктах, которые в свою очередь используются в *web*-разработке. Так как язык является интерпретируемым, то производительность и безопасность зависит от конкретной реализации интерпретатора («движка»).

Язык программирования *Java*.

Java – сильно типизированный объектно-ориентированный язык программирования [6]. В настоящее время разработкой занимается компания *Oracle*. Основной особенностью данного языка является то, что программы на *Java* транслируются в байт-код, который выполняется на виртуальной машине *Java*. Виртуальная машина *Java* в свою очередь является интерпретатором и передает инструкции оборудованию, что обеспечивает легкую переносимость программы на разные платформы. В настоящее время *Java* является одним из самых популярных языков программирования.

Выбор языка программирования.

В качестве языка программирования был выбран язык *Java*. В отличие от программ написанных на *Python* программы на *Java* обладают большей производительностью, по сравнению с языком *Go* *Java* давно используется программистами, имеет большое сообщество, хорошую документацию, множество популярных фреймворков. *JavaScript* проигрывает *Java* в безопасности [7-9].

1.2.2.4 Анализ фреймворков

Framework - это план, который разработчики используют для создания продукта.

Мы можем сравнить его с домом. Если вы хотите построить его сами, вы можете купить материалы и сделать его самостоятельно, смазав свои собственные ладони. Вы также можете скачать готовую модель из Интернета. Конечно, у вас могут возникнуть проблемы с обоями и мебелью. Но, по крайней мере,

фундамент, стены, крыша и дренажная система на месте. Потому что фреймворк описывает, как они строятся. Фреймворки делают программирование быстрым, простым и эффективным, используя готовые фрагменты кода, так что вам не придется писать их с нуля. Выбор фреймворка зависит от потребностей каждого проекта. *Technotification* собрала 10 фреймворков для Java-программистов и разработчиков.

Spring Framework - один из самых популярных Java-фреймворков. Большинство разработчиков предпочитают этот язык, поскольку он позволяет реализовать зависимости.

Spring идеально подходит для создания бизнес-приложений и конфигурационных моделей на базе *Java*. Это позволяет разработчикам сосредоточиться на бизнес-логике приложения. Он тесно интегрирован с фреймворком *Spring Boot*.

PrimeFaces - один из лучших легких *Java*-фреймворков. Его можно загрузить в виде отдельного файла *.jar*. *PrimeFaces* используется уже несколько лет и является интерфейсной основой для спецификаций *JavaServer Faces* и *Java EE*. Он также включает более 100 компонентов, наиболее важными из которых являются валидация на стороне клиента и набор инструментов для сенсорных смартфонов.

Blade - это легкий фреймворк *MVC*, основанный на *Java 8*. Он прост и включает в себя *RESTful* интерфейс маршрутизации. Это также один из немногих *Java*-фреймворков без раздражающих блокировщиков. Под "легким" мы подразумеваем небольшой исходный код, не более 500 килобайт. Чтобы использовать *Blade*, необходимо создать типичный проект *Maven*. Фреймворк поддерживает модульность *Java 9* и большое количество веб-компонентов *Java*.

Dropwizard - это мощный *Java*-фреймворк, оптимизированный для разработки *RESTful* сервисов. Он также идеально подходит для написания микросервисов на этом языке и обеспечивает легкий доступ к мощным библиотекам *Java*, таким как *Google Guava*, *Jetty Server*, *Hibernate Validator*, *Logback*, *Joda Time* и *Jackson* для работы с файлами *JSON*. *JSON* устраняет

необходимость написания кода для метрик и конфигурации, поэтому вы можете сосредоточиться на функциональности приложения.

Dropwizard поможет вам достичь максимальной эффективности разработки. Четкая документация для начинающих также является большим подспорьем.

Google Web Toolkit (GWT) - это фреймворк, выпущенный компанией *Google*, чтобы помочь разработчикам писать веб-приложения на *Java*. Он позволяет разработчикам писать код на *Java* и компилировать его в *JavaScript* для использования в веб-браузерах.

GWT поддерживается командой опытных программистов *Google*. С помощью этого фреймворка вы можете создавать сложные веб-приложения без практического опыта использования внешних языков разработки, таких как *JavaScript*.

Фреймворк включает в себя ряд уникальных особенностей, таких как абстракция интерфейса, кросс-браузерная совместимость и интернационализация.

Фреймворк *JavaServer Faces MVC* была представлена 14 лет назад. Он упрощает разработку пользовательских интерфейсов для веб-приложений. Важнейшей особенностью этого фреймворка является то, что созданные в нем компоненты пользовательского интерфейса могут быть повторно использованы на других сайтах. *JSF* использует *Facelets* в качестве шаблона.

JHipster - относительно молодой фреймворк, выпущенный в 2013 году. Он объединяет *Spring Boot*, *Angular* и *React* в единый фреймворк. Создать современное веб-приложение на *Java* очень просто.

Интегрировав *Spring Boot*, вы сможете создавать приложения на основе фреймворка *Spring*. *JHipster* использует *Angular*, *React* и *Bootstrap*. Кроме того, *JHipster* предлагает два типа архитектуры: монолитная или микросервисы. В первом случае *front-end* и *back-end* реализованы в одном приложении, а во втором - по отдельности.

Spark - лучший выбор для программистов, разрабатывающих веб-приложения на *Java*. С его помощью можно быстро и легко создавать веб-основы. *Spark* поддерживает почти все функции *Java 8* и имеет превосходный *API*.

1.2.2.5 Описание *Swagger*

"*Swagger*" - это набор инструментов, позволяющий автоматически комментировать *API*, интерфейсов связи между различными программными продуктами, которые всегда присутствуют в каждом проекте. *Swagger* автоматически генерирует документацию, чтобы облегчить понимание *API* компьютерами и людьми.

Swagger автоматически генерирует документацию в виде файлов JSON на основе кода и правил, которые могут быть встроены в веб-сайт или приложение и отправлены клиентам, позволяя пользователям интерактивно просматривать документацию. Создание такой разметки происходит гораздо быстрее, чем написание документа с нуля.

Иногда его называют "идеальным шаблоном". Фреймворк - это набор инструментов и правил, из которых состоит *Swagger*. Некоторые из этих инструментов бесплатны и имеют открытый исходный код, другие предоставляются компаниями за плату.

Общепринятое название «Сваггер», но есть другое название - *OpenAPI*, название спецификации, с которой работает *Swagger*, но иногда используется для описания продукта.

Swagger используют разработчики, которые хотят быстро документировать *API* проекта и не имеют времени на написание документации с нуля. Разработчики, которые хотят внедрить *API* сторонних разработчиков, нужна документация, в том числе созданная в *Swagger*. Технические писатели, составляющие документацию для проектов в качестве основного или дополнительного инструмента.

Предназначение *Swagger*.

Основная цель *Swagger* - автоматическая генерация документации, понятной как людям, так и машинам. Поэтому он в основном используется для быстрого и простого документирования кода *API*. Он часто используется в сочетании с архитектурой *RESTful API*.

Разработка *API*.

Swagger используется, когда разработчику нужна документация для разработки продукта или когда он хочет создать документацию из кода. Иногда с *Codegen Swagger* возможно и обратное - генерация кода из документации. Подробнее об этом позже.

Интеграция API.

Опции генерации кода используются для подключения к *API* других проектов. *Codegen swagger* может генерировать код для взаимодействия конечного пользователя с *API*, экономя время.

Способы создания документации.

На основе кода. Чтение кода *API* и создание на его основе документации. Этот метод считается более простым, так как разработчику не нужно знать спецификации и не нужно писать ничего, кроме кода. Однако, поскольку второй метод дает более подробное и ясное объяснение, рекомендуется использовать его только в случае крайней необходимости документирования.

Согласно спецификации. Второй способ заключается в использовании спецификации *Swagger*, называемой *OpenAPI*. Это сложнее, поскольку вам необходимо знать официальный язык правил - вам нужно описать единицы кода на этом языке, чтобы инструмент понимал, что вы написали, и создавал документацию. Однако такой подход является более точным, поскольку этот вид документации более читабелен и доступен для восприятия человеком. Вы должны писать в *JSON*, *YAML* или в редакторе *Swagger*, описанном ниже.

Спецификация *Swagger*.

Спецификация - это набор правил, которые описывают, как должна выглядеть и работать технология. Это теоретический "скелет" для проектирования будущего программного обеспечения. На основе этого пишется код - приложение, которое работает в соответствии с логикой, описанной в спецификации. Таким образом, *OpenAPI* - это набор правил и стандартов для определения *API*.

Это работает, когда эксперт пишет код, используя стандартные правила работы с текстом. *Swagger* "понимает" то, что мы пишем, и превращает это в

интерактивную, человекочитаемую документацию.

Организация спецификаций.

Спецификация содержит базовые единицы, которые состоят из более мелких единиц. Они используются для описания документации. Синтаксис следующий:

<название объекта>: <значение>

или

<название объекта>:

<название вложенного объекта>: <значение>

<название вложенного объекта>: <значение>

И так далее.

Объект может быть, например, именем пользователя, именем устройства, ссылкой или чем-то еще. Существует восемь основных объектов; остальные объекты являются вложенными объектами:

Openapi. Его значение - это версия *OpenAPI*, используемая в проекте,

Info. Он содержит подобъекты, которые содержат основную информацию об *API*: название, описание, лицензию, информацию о разработчике и т.д.,

Servers. Он содержит ссылки на серверы, представляющие основные внешние пути, за исключением конечных точек,

Paths. Указывает конечные точки или конечные точки, которые являются концом пути к объекту. Каждая конечная точка связана с запросом *GET*, *POST*, *DELETE* или *PUT*, используемым для связи с устройством,

Components. В нем хранятся схемы, которые могут быть использованы в различных местах документации. Например, вы можете создать схему "Пользователь" с полями "Имя", "Адрес" и другими. Как только схема определена в элементах, ее можно использовать в коде документации,

Tags. хранит метаданные о тегах: название, описание и так далее,

Security. Описывает методы безопасности, которые могут быть

использованы в API,

ExternalDocs. Содержит ссылки на внешние документы, часто с дополнительной информацией. Это может включать, например

Преимущества использования *Swagger*:

- экономит время программистов и технических писателей, упрощая и ускоряя документирование;
- работу можно сократить и передать генерацию кода инструменту *Swagger codegen*: это также экономит время;
- документация является исчерпывающей и понятной для всех читателей: инженеров, обычных пользователей и роботов;
- очень гибкий, документация может генерироваться из кода или внутри самого кода, используя визуальный интерфейс для тестирования и быстрой навигации;
- может быть легко интегрирована в веб-сайты и полноценные интерактивные приложения. Однако лучше всего разработать свой собственный дизайн. Стандартный пользовательский интерфейс не является лучшим для создания веб-сайтов.

Недостатки

Некоторые разработчики не любят *Swagger* и считают его ненужным или избыточным, поскольку он усложняет *API*. Отношения между внутренними и внешними страницами не очень понятны. Это действительно может быть правдой - *Swagger* генерирует много кода, который не является проблемой для одних разработчиков, но сложен для понимания других. Другая проблема заключается в том, что документация не совсем понятна.

Еще одна критика *Swagger* - отсутствие подробных пояснений; многие люди используют для документации такие инструменты, как *Markdown*. Их можно использовать, чтобы выделить важные разделы, сделать сноски в разных частях документа, добавить ссылки и т.д. *Swagger* отличается: документация написана так, чтобы ее было легко понять. *Swagger* отличается: документация, создаваемая *Swagger*, представляет собой файл *JSON*, а текст в этом формате не

может быть аннотирован.

1.3 Реализация

1.3.1 Структура БД

При разработке сервиса учёт спортивных команд была использована СУБД *PostgreSQL*. Далее представлено описание структуры разработанной БД. Таблица 1 содержит описание структуры таблицы «Игроки (*app_player*)».

Таблица 1 — Структура таблицы «Игроки (*app_player*)»

Название поля	Описание поля	Тип данных
<i>id</i>	Идентификатор игрока	<i>VarChar(255)</i>
<i>first_data</i>	Дата вступления	<i>VarChar(255)</i>
<i>last_data</i>	Дата конца	<i>VarChar(255)</i>
<i>name</i>	Наименование игрока	<i>VarChar(255)</i>
<i>sport_type</i>	Вида спорта	<i>VarChar(255)</i>
<i>team</i>	Команда	<i>VarChar(255)</i>

Таблица 2 содержит описание структуры таблицы «Команда (*app_team*)».

Таблица 2 — Структура таблицы «Команда (*app_team*)»

Название поля	Описание поля	Тип данных
<i>id</i>	Идентификатор команды	<i>VarChar(255)</i>
<i>name</i>	Наименование команды	<i>VarChar(255)</i>
<i>sport_type_id</i>	Идентификатор вида спорта	<i>VarChar(255)</i>

Таблица 3 содержит описание структуры таблицы «Вид спорта (*app_sport_type*)».

Таблица 4 содержит описание структуры таблицы «История изменений (*app_history*)».

Таблица 3 — Структура таблицы «Вид спорта (*app_sport_type*)»

Название поля	Описание поля	Тип данных
<i>id</i>	Идентификатор вида спорта	<i>VarChar(255)</i>
<i>name</i>	Наименование вида спорта	<i>VarChar(255)</i>

Таблица 4 — Структура таблицы «История изменений (*app_history*)»

Название поля	Описание поля	Тип данных
<i>id</i>	Идентификатор игрока	<i>VarChar(255)</i>
<i>first_data</i>	Дата вступления	<i>VarChar(255)</i>
<i>last_data</i>	Дата конца	<i>VarChar(255)</i>
<i>name</i>	Наименование игрока	<i>VarChar(255)</i>
<i>sport_type_id</i>	Идентификатор вида спорта	<i>VarChar(255)</i>
<i>team_id</i>	Идентификатор команды	<i>VarChar(255)</i>

1.3.2 Описание *REST API*

Разработка *API* один из важнейших этапов создания микросервиса, так как от полноты реализации функционала *API* зависит удобство реализации остальных микросервисов и производительность приложения. *API*, описанное в данной работе, соответствует архитектурному стилю *REST* и поддерживает следующие функции:

- создание игрока;
- обновление записи об игроке;
- удаление игрока;
- получение записи об игроке по идентификатору/датам вступления/имени;
- поиск игроков;
- создание команды;
- обновление записи о команде;
- удаление команды;

- получение записи о команде по идентификатору;
- поиск команды;
- создание спортивного типа;
- обновление записи о спортивном типе;
- удаление спортивного типа;
- получение записи о спортивном типе по идентификатору;
- поиск спортивного типа;

Для примера рассмотрим запрос на создание игрока.

Спецификация *REST API* требует того, чтобы при операциях создания использовался *HTTP* метод *POST*. Также представляются требования к формированию *URI* запроса, например для данного запроса *URI* имеет следующую структуру: название микросервиса/название сущности/операция (или идентификатор объекта). Тело запроса состоит из полей, которые необходимы для создания пользователя. При успешном создании пользователя возвращается код 200 и объект, описывающий созданного пользователя (который включает в себя поля, переданные при создании и новые поля, добавленные после создания). В случае ошибки выдается код 400 и сообщение «*Bad Request*».

Листинг 1. Запрос на создание пользователя:

Метод: *POST*

URI: */player*

Тело запроса:

```
{
  "firstData": "string",
  "id": 0,
  "lastData":
  "string",
  "name": "string",
  "sportTypeId": 0,
  "teamId": 0
}
```

Ответ:

```
200    {
        "firstData": "string",
        "id": 0,
        "lastData":
        "string",
        "name": "string",
        "sportTypeId": 0,
        "teamId": 0
    }

400    Bad Request
        { "errorMessage": "string" }
```

1.3.3 Описание классов

Основываясь на описанных ранее СУБД и *REST API*, для удобства взаимодействия архитектура разбита на следующие функциональные блоки:

- классы-сервисы, обеспечивают взаимодействие контроллеров с репозиториями, вмещают в себя общую массу бизнеслогики;
- классы-репозитории, отвечают за взаимодействие СУБД с сущностями, передаются из классов-сервисов соответственно;
- классы-сущности, описывают сущности БД в программном коде, помогают во взаимодействии с данными, полученными из классов-репозитория;
- классы-контроллеры, выполняющие запросы *REST API*, валидируют полученные данные, возвращают ответ.

В пример рассмотрим сущность «Игрок» для более наглядного примера взаимодействия описанных выше классов.

Представленный класс сущность описывает структуры «Таблица», а так же описывает остальные сущностями («Команда», «Вид спорта», «История»). В классе описаны методы, которые позволяют записывать различные данные (имя, дата вступления, дата выхода, вид спорта, команда) в определённые поля. Если брать во внимание поля «Вид спорта» и «Команда», то при запросе устанавливается идентификатор, который в свою очередь проверяет

существование записи о том или ином спорте/команде и переносит данные в уже существующую таблицу. Устанавливает отношения между сущностями («*ManyToOne*», «*ManyToOne*»), после создания сущности автоматически создаётся запись в истории изменений (таблица). *UML* диаграмма классов сущности «Игрок» представлена в приложении А.2.

Класс «*PlayerController*» является классом-контроллером. Основной задачей является обработка входного запроса, валидация, обработка данных («*PlayerService*»), и возврат полноценного ответа в формате *JSON*.

UML диаграмма, описывающая структуру класса-контроллера «*PlayerController*», представлена в приложении А.2.

Класс «*PlayerRepo*» является классом-репозиторием. Реализует класс *JpaReository*, наследуя из него стандартные запросы. Задачей является работа с СУБД и осуществление операций создания, обновления, удаления и поиска пользователей. Допустим, операция поиска представляет собой набор специфичных для каждого запроса, обрабатываемого контроллером («*PlayerController*») функций, которые ищут игрока на основе ограничений, передаваемых при запросе или установленных в самих сущностях.

UML диаграмма, описывающая структуру класса-репозитория «*PlayerRepo*», представлена в приложении А.2.

Класс «*PlayerService*» является классом-сервисом. Функции представленного класса позволяют работать с объектами, описывающими игроков, подготавливая объекты к передачи для обработке в репозитории.

UML диаграмма, описывающая структуру класса-сервиса «*PlayerService*», представлена в приложении А.2.

1.3.4 Программная реализация

Как упоминалось в предыдущем разделе, для реализации данного микросервиса используется язык программирования *Java* и фреймворк *Spring*. Был реализован *API* для работы с игроками, командами и видами спорта, а также функции для работы с ним.

На *UML* диаграмме (Приложение А.2) показана последовательность

прелесть заключается в том, что он дает возможность не только интерактивно просматривать спецификацию, но и отправлять запросы – так называемый *Swagger UI*

Структура базы данных приложения подразумевает, что игроки не могут существовать в отрыве от команды и спортивного, поэтому перед созданием первого пользователя необходимо создать спортивный тип и команду, в которую будет добавлен игрок. Для этого можно использовать запрос «*POST /team*», рисунки 7-10 и «*POST /sport_type*» в тело которого добавляется описание создаваемой организации в формате *JSON*. При успешном выполнении запроса возвращается объект с описанием организации, в котором особое внимание стоит уделить полю «*id*». Значение этого поля необходимо указать в запросе создания пользователя в поле «*sportTypeId*», чтобы привязать игрока к команде, статус «200» сигнализирует об успешном выполнении запроса. Результат выполнения запроса на создание команды представлен на рисунках 12-14.

Операции с спортивными типами Sport Type Controller

GET /sport-type Поиск всех записей спортивных типов

POST /sport-type Создание записи спортивного типа

Parameters Try it out

Name	Description
req * required object (body)	req Example Value Model <pre>{ "name": "string" }</pre> Parameter content type application/json

Responses Response content type */*

Code	Description
201	Created
401	Unauthorized
403	Forbidden
404	Not Found

Рисунок 7 — Окно Swagger при работе с созданием вида спорта

POST /sport-type Создание записи спортивного типа

Parameters Cancel

Name	Description
req required object (body)	req Edit Value Model

```
{
  "name": "Футбол"
}
```

Cancel

Parameter content type
application/json

Execute

Рисунок 8 — Тело запроса на «Создание вида спорта» с заполненными полями

Responses Response content type */*

Curl

```
curl -X POST "http://localhost:8181/sport-type" -H "accept: */*" -H "Content-Type: application/json" -d '{"name": "Футбол"}"
```

Request URL

```
http://localhost:8181/sport-type
```

Server response

Code	Details
201	Response headers <pre>connection: keep-alive content-length: 0 date: Mon06 Jun 2022 15:06:15 GMT keep-alive: timeout=60</pre>

Responses

Code	Description
201	Created
401	Unauthorized
403	Forbidden
404	Not Found

Рисунок 9 — Ответ на отправленный запрос об успешном выполнении

	id	name
1	ea543807-cc1e-47a0-9595-e6aa8da9415a	Хоккей

Рисунок 10 — Содержание базы данных «sport_type» до запроса

WHERE		ORDER BY	
	id		name
1	236ee901-69df-4383-b1f5-cfa376bb4581		Хоккей
2	71cfb055-c60b-43bf-a261-84850eef61b2		Футбол

Рисунок 11 — Содержание базы данных «sport_type» с полученными данными

Далее добавим игрока в ранее созданную нами команду «Алмаз», для этого формируем запрос «POST /player», заполняем тело запроса нужными полями, представленными на рисунке 15, результат работы на рисунках 16, 17.

POST /team Создание записи о команде

Parameters Cancel

Name	Description
req required	req
object	
(body)	Edit Value Model

```
{
  "name": "Алмаз",
  "sportTypeId": "ea543807-cc1e-47a0-9595-e6aa8da9415a"
}
```

Cancel

Parameter content type
application/json

Рисунок 12 — Тело запроса на «Создание команды» с заполненными полями

Request URL
http://localhost:8181/team

Server response

Code	Details
200	<p>Response headers</p> <pre>connection: keep-alive content-length: 0 date: Mon 06 Jun 2022 15:21:52 GMT keep-alive: timeout=60</pre>

Responses

Code	Description
------	-------------

Рисунок 13 — Результат выполнения запроса «Создание команды» в Swagger

	id	name	sport_type
1	b95992e7-0647-49be-b232-c3add5ff37b3	Алмаз	ea543807-cc1e-47a0-9595-e6aa8da9415a

Рисунок 14 — Содержание базы данных «team» с полученными данными

POST /player Создание записи об игроке

Parameters Cancel

Name	Description
req ★ required	req
object	
(body)	Edit Value Model

```
{
  "firstData": "30-01-2000",
  "lastData": "Настоящее время",
  "name": "Иванов Иван Иванович",
  "sportTypeId": "ea543807-cc1e-47a0-9595-e6aa8da9415a",
  "teamId": "b95992e7-0647-49be-b232-c3add5ff37b3"
}
```

Рисунок 15 — Тело запроса на «Создание игрока» с заполненными полями

Responses Response content type: */*

Curl

```
curl -X POST "http://localhost:8181/player" -H "accept: */*" -H "Content-Type: application/json" -d '{"firstData": "30-01-2000", "lastData": "Настоящее время", "name": "Иванов Иван Иванович", "sportTypeId": "ea543807-cc1e-47a0-9595-e6aa8da9415a", "teamId": "b95992e7-0647-49be-b232-c3add5ff37b3"}'
```

Request URL

```
http://localhost:8181/player
```

Server response

Code	Details
200	<p>Response headers</p> <pre>connection: keep-alive content-length: 0 date: Mon06 Jun 2022 15:27:24 GMT keep-alive: timeout=60</pre>

Responses

Code	Description
200	OK
201	Created
401	Unauthorized

Рисунок 16 — Результат выполнение запроса «Создание игрока»

id	first_data	last_data	name	sport_type	team
1	a7648c8d-32e3-4ddb-af69-2b5bad4cba07	30-01-2000	Настоящее время	Иванов Иван Иванович	ea543807-cc1e-47a0-9595-e6aa8da9415a
					b95992e7-0647-49be-b232-c3add5ff37b3

Рисунок 17 — Содержание базы данных «player» с полученными данными

Чтобы получить данные из базы данных существует запрос «GET /player/{id}», данный запрос осуществляет поиск по идентификатору сущности, результат работы представлен на рисунке 18.

В соответствии с рисунками 17, 14, 11, в ответ на запрос пришли верные данные об игроке, команде, в которой состоит и виде спорта, в котором выступает.

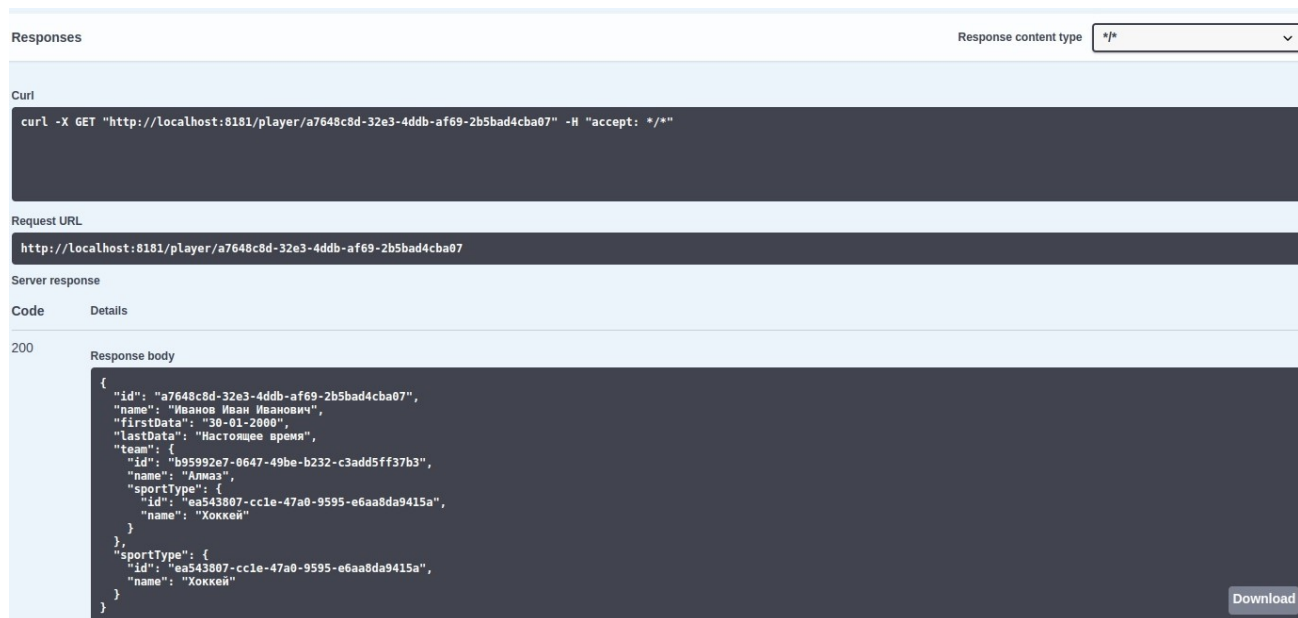


Рисунок 18 — Результат выполнения запроса «Поиск игрока по идентификатору»

Для изменения данных нужно указать идентификатор игрока и поля, которые нам нужно поменять, ответ на запрос код «200», рисунок 19.



Рисунок 19 — Изменение данных

2 Безопасность жизнедеятельности

2.1 Расчет эквивалентного уровня звука на рабочих местах в компьютерных залах, офисах и других помещениях, где установлены ПЭВМ

В гигиенической практике шумом называют всякий нежелательный для человека звук. Под его воздействием увеличивается утомляемость, ухудшается восприятие звуковых сигналов и разборчивость речи. У человека может возникать головокружение при выполнении всех видов работ или затрудненное зрение; возникают различные профессиональные заболевания — тугоухость, гипертония.

Любое физическое явление представляет собой распространяющееся механическое колебательное движение части упругой среды с малыми амплитудами. Источником его является любое колеблющееся тело, выведенное из устойчивого состояния какой-либо внешней силой и перемещавшееся вокруг своего положения равновесия в периоды циклического разряжения или уплотнения. Этот процесс может распространяться последовательно на смежные частицы в виде волны λ .

$$\lambda = \frac{c}{f} = cT, \quad (1)$$

где c – скорость звука в среде,

f – частота колебаний,

T – период колебаний.

При распространении звуковой волны происходит перенос энергии в пространстве, называемом звуковым полем. Общее количество энергии, которое источник звука излучает в окружающее пространство называется звуковой мощностью источника (W). Реальные звуковые мощности некоторых источников звука характеризуют следующие приблизительные цифры (речь идет о порядке величин):

- шепот – 10^{-9} Вт;

- обычный разговор – 10^{-5} Вт;

Изм.	Лист	№ докум.	Подпись	Дата

- крик – 10^{-3} Вт;
- цепная пила по дереву – 1 Вт;
- большой оркестр – 10 Вт;
- 4 турбореактивный самолетный двигатель – 10^4 Вт;
- стартовый двигатель мощной космической ракеты – 10^8 Вт.

При оценке шума в какой-либо точке звукового поля (например, на рабочем месте) интерес представляет не общая акустическая мощность источника звука, и даже его часть. Часть общей мощности источника шума, приходящаяся на единицу площади заданного поля и расположенной перпендикулярно распространяющейся звуковой волне называется интенсивностью звука (J).

Непосредственное измерение интенсивности звука связано с большими техническими трудностями, и в настоящее время нет приборов для непосредственного измерения этого параметра. При помощи акустических приборов (например, шумомеров) можно измерить звуковое давление (P) в зависимости от интенсивности звука (J):

$$J = \frac{P^2}{\rho a}, \quad (2)$$

где ρ – плотность среды,

a – скорость звука в среде.

Звуковым давлением называется разность между мгновенным значением полного давления в какой-либо точке звукового поля и средним давлением, которое наблюдается при невозмущенной среде («атмосферное давление»). Поскольку в течение полного колебательного цикла звуковое давление меняется от нуля до отрицательного максимума, а затем вновь возвращается к нулю и т.д. (в пределах полной цикличности):

$$P = \sqrt{P^2} = \sqrt{\frac{1}{T} \int_0^T P^2(\tau) d\tau}, \quad (3)$$

где P – означает осреднение во времени,

T – продолжительность цикла.

В органе слуха человека такое осреднение происходит примерно за 30 —

100 мс и воспринимается человеком как специфичный звуковой сигнал, если частоту колебаний в диапазонах 16 Гц — 20 кГц. Колебания с частотой менее 16 Гц и более 20 кГц называются инфразвуком. Интенсивность звуков и частота шума, с которыми приходится иметь дело на практике (как в теории), могут изменяться или меняться по бесконечному диапазону. В этих условиях очень удобно использовать логарифмическую шкалу, которая позволяет значительно уменьшить диапазон измеряемых величин. Применению логарифмической шкалы в акустике способствовало обстоятельство, что громкость звуков при оценке на слух увеличивается примерно пропорционально его коэффициенту интенсивности или звука.

Основными характеристиками шума являются уровни звукового давления и уровни интенсивности звука, определяемые по формулам (2,3), дБ,

$$L_p = 20 \lg \frac{J}{J_0}.$$

$$L_p = 20 \lg \frac{P}{P_0},$$

где P_0 и J_0 – пороговые (опорные) значения звукового давления и интенсивности шума на частоте 1000 Гц. Они стандартизованы и равны соответственно $2 \cdot 10^{-5}$ Па и 10^{-12} Вт/м². При нормальных атмосферных условиях $L_p = L_J$, что очень удобно при расчетах и акустических измерениях. При условиях, отличных от нормальных, связь между L_p и L_J определяется выражением:

$$L_J = L_p + 10 \lg \frac{p_0 c_0}{p c},$$

где p_0 и c_0 – плотность воздуха и скорость звука при нормальных атмосферных условиях; p и c – то же при условиях, отличных от нормальных. В производственном помещении обычно бывает несколько источников шума. Суммарный уровень звукового давления нескольких различных источников звука определяется по формуле, дБ,

$$L = 10 \lg [10^{(L_1/10)} + 10^{(L_2/10)} + \dots + 10^{(L_n/10)}],$$

где $L_1, L_2 \dots L_n$ – уровни звукового давления, создаваемые каждым из источников

звука в исследуемой точке пространства.

Суммарный уровень шума от одинаковых по своему уровню источников определяется по формуле

$$L_{\Sigma} = L_i + 10 \lg n,$$

где L_i – уровень звукового давления одного источника, дБ;

n – количество источников шума.

Например, два одинаковых источника совместно создадут уровень на 3 дБ больше, чем каждый источник. Суммарный уровень шума от двух различных по своему уровню источников можно определить по формуле

$$L_{\Sigma} = L_{\max} + \Delta L,$$

где $L_{\max}$ – максимальный уровень звукового давления одного из двух источников,

ΔL – поправка, зависящая от разности между максимальным и минимальным уровнем звукового давления в соответствии с приведенными ниже данными.

В результате сравнения, полученного по данным этой таблицы можно определить суммарный уровень звукового давления нескольких различных источников звука. По разности двух уровней L_1 и L_2 по таблице 5 определяют добавку щёлкнувшей к большему уровню L_1 . Уровень $L_{1,2}$ суммируется с уровнем L_3 и получают уровень $L_{\text{сум}}$.

Метод расчета применяется в тех случаях, когда есть данные об уровне и продолжительности воздействия шума на рабочем месте или различных помещениях. Эквивалентный уровень звука рассчитывается с использованием поправок на время действия каждого уровня звука, определяемых по таблице 6.

Расчет производится следующим образом. В каждом измеренном уровне звука добавляется поправка в таблице, соответствующая его времени действия (в часах или процентах от общего срока). Затем полученные уровни звука складываются в соответствии с описанным выше.

Пример расчета.

					ПГУ1 090301.04.001 ПЗ	Лист
						40
Изм.	Лист	№ докум.	Подпись	Дата		

Уровни шума за 8-часовую рабочую смену составляли 80, 86 и 94 дБА в течение 5, 2 и 1 ч соответственно. Этим значениям времени соответствуют поправки из предыдущей таблицы, равные -2, -6, -9 дБ. Складывая их с уровнями шума, получаем 78, 80, 85 дБА. Теперь, используя таблицу 2 значений поправок ΔL , складываем эти уровни попарно:

$$L_{1,2} = L_2 + \Delta L = 80 + 2,2 = 82 \text{ дБА.}$$

Таблица 5 — Значение поправки ΔL при сложении уровней шума

Разность слагаемых уровней $L_1 - L_2$, дБ	Добавка ΔL , прибавляемая к большому из уровней L_1 , дБ
0	3,0
1	2,5
2	2,2
3	1,8
4	1,5
5	1,2
6	1,0
7	0,8
8	0,6
10	0,4

$$L_{1,2,3} = L_3 + \Delta L = 85 + 1,8 = 86,8 \text{ дБА.}$$

Округляя, получаем окончательный эквивалентный уровень шума 87 дБА. При исследовании шумов можно обнаружить, что воздействие этих звуков равно воздействию шума с постоянным уровнем 87 дБА в течение 8 ч. При исследовании шумов обычно весь слышимый диапазон звуковых колебаний по частоте разбивается на отдельные полосы, каждая из которых характеризуется граничными частотами: нижней (f_n), верхней (f_v) и средней (f_{cp}). За среднюю частоту полосы принято принимать среднегеометрическую частоты, которая определяется по формуле:

$$f_{\text{ср}} = \sqrt{f_{\text{н}} f_{\text{в}}}.$$

Таблица 6 — Значение поправок на время действия шума

Поправка в дБ	Время	
	в часах	в %
0	8	100
-0,6	7	88
-1,2	6	75
-2	5	62
-3	4	50
-4,2	3	38
-6	2	25
-9	1	12
-12	0,5	6
-15	15	3
-20	5	1

Чаще всего применяются октавные и третьоктавные полосы. Октавная полоса (октава) – это такая частотная полоса, в которой верхняя граничная 9 частота ($f_{\text{в}}$) в два раза больше нижней частоты ($f_{\text{н}}$). В третьоктавной полосе это соотношение равно 1,26. При гигиенической оценке шума и его нормировании акустический диапазон частот разделяют на девять октавных полос со среднегеометрическими частотами 31,5; 63; 125; 250; 500; 1000; 2000; 4000 и 8000 Гц. Граничные частоты для этих полос соответственно равны 22–44 Гц., 44–88 Гц., 88–177 Гц., 177–355 Гц., 355–710 Гц., 710–1420 Гц., 1420–2840 Гц., 2840–5680 Гц., 5680–11360 Гц.

В качестве одночисловой характеристики шума применяется оценка уровня звука в децибелах (А), получаемая посредством измерения звукового сигнала на характеристике «А» чувствительности шумомера. С помощью специальных

фильтров характеристика «А» чувствительности шумомеров подобрана таким образом, чтобы между субъективной реакцией человека и уровнем звукового давления существовало соответствие. Иными словами, характеристика «А» шумомеров хорошо имитирует чувствительность человеческого уха во всем звуковом диапазоне.

					ПГУ1 090301.04.001 ПЗ	Лист
						43
Изм.	Лист	№ докум.	Подпись	Дата		

Заключение

В ходе выполнения данной работы была изучена методология проектирования приложения с микросервисной архитектурой и приобретены навыки работы с фрейворком *Spring*. При реализации поставленной задачи были получены следующие результаты:

- разработана структура базы данных для СУБД *PostgreSQL*;
- разработан прикладной интерфейс для взаимодействия с остальными микросервисами приложения (*REST API*);
- разработано и отлажено приложение «Микросервис учёта спортивных команд при проведении турниров».

Сервис поддерживает полный функционал для интеграции с остальными микросервисами спортивного ресурса.

					ПГУ1 090301.04.001 ПЗ	Лист
						44
Изм.	Лист	№ докум.	Подпись	Дата		

Список использованных источников

1. Пащенко Д. В. и др. Моделирование автоматизированной информационной системы государственных услуг //XXI век: итоги прошлого и проблемы настоящего плюс. – 2018. – №.
2. Пащенко Д. В. и др. ПРОГРАММНО-ИНФОРМАЦИОННАЯ СИСТЕМА ДЛЯ АНАЛИЗА ПУБЛИКАЦИЙ ПОЛЬЗОВАТЕЛЕЙ В СОЦИАЛЬНЫХ СЕТЯХ //XXI век: итоги прошлого и проблемы настоящего плюс. – 2019. – Т. 8. – №. 2. – С. 14-21.
3. Ньюмен С. Создание микросервисов = Building Microservices //Питер.— 2016. — с. 56-246.
4. Дж. Уорсли, Дж. Дрейк. PostgreSQL для профессионалов //Питер.— 2003.— с. 23-320.
5. Николай Прохоренок, Владимир Дронов. HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера //БХВ-Петербург.— 2015.— с. 20-348.
6. Шилдт Г. Java 8: руководство для начинающих //Вильямс.— 2018.— с. 149- 348.
7. Марк Саммерфилд. Python на практике //ДМК Пресс.— 2014.— с. 120-310.
8. Марк Саммерфильд. Программирование на Go. Разработка приложений XXI века. //ДМК Пресс.— 2016.— с. 34-251.
9. Эрик Фримен, Элизабет Робсон. Изучаем программирование на JavaScript //Питер.— 2018.— с.125-340.

Приложение А

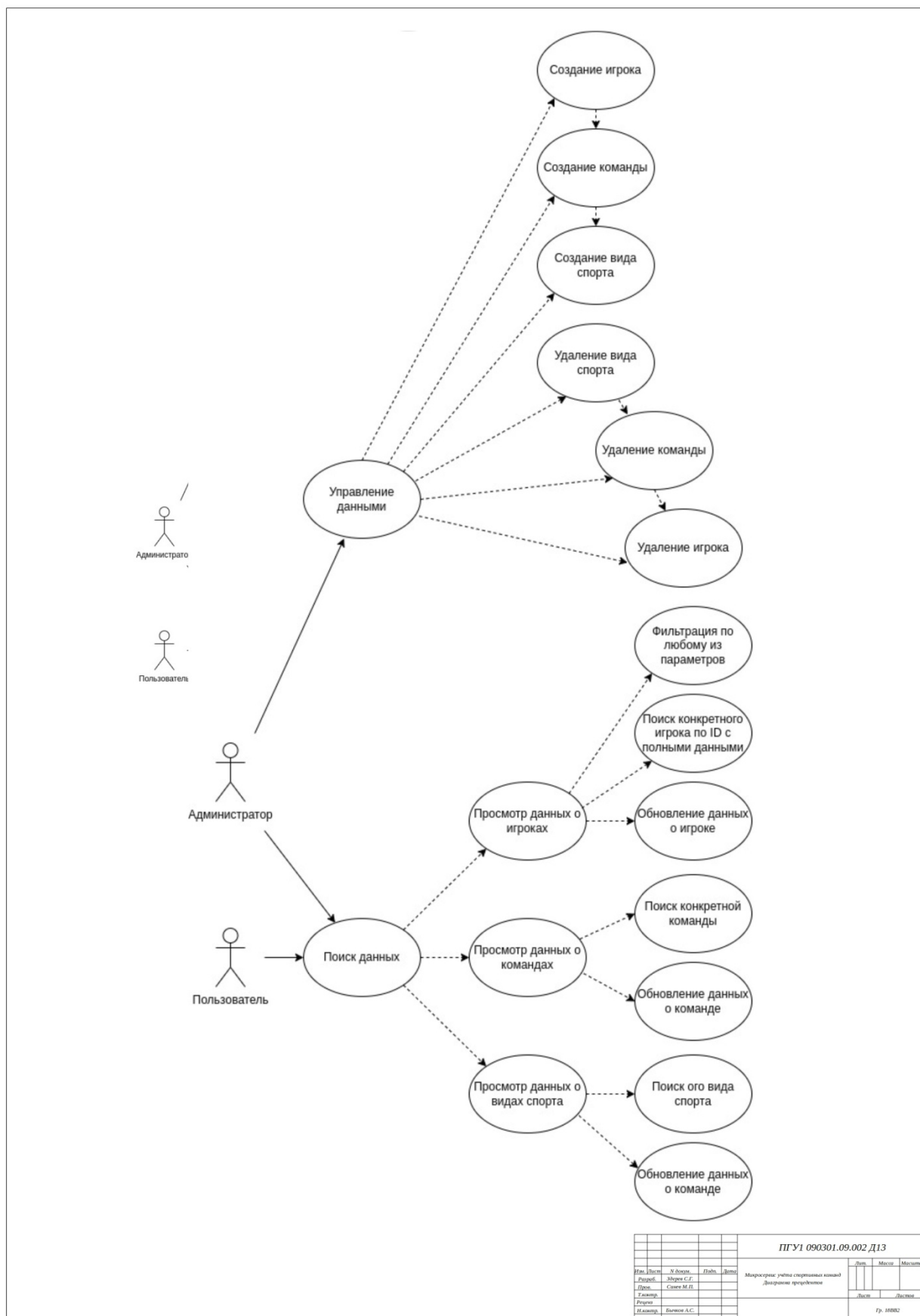
(обязательное)

Микросервис учета спортивных команд при проведении турниров

UML-диаграммы

					ПГУ1 090301.04.001 ПЗ	Лист
						46
Изм.	Лист	№ докум.	Подпись	Дата		

Приложение А.1 – Диаграмма прецедентов



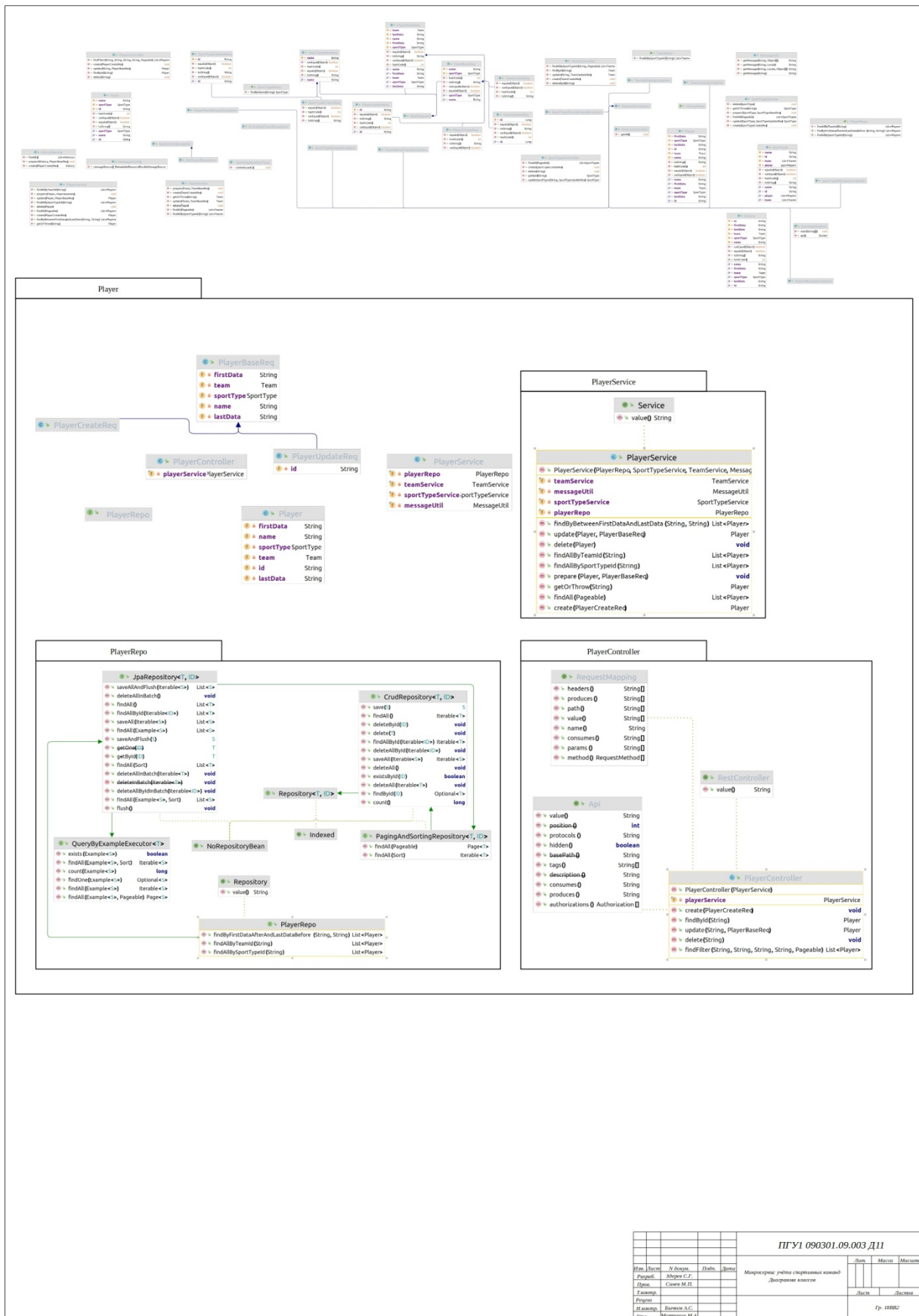
Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1 090301.04.001 ПЗ

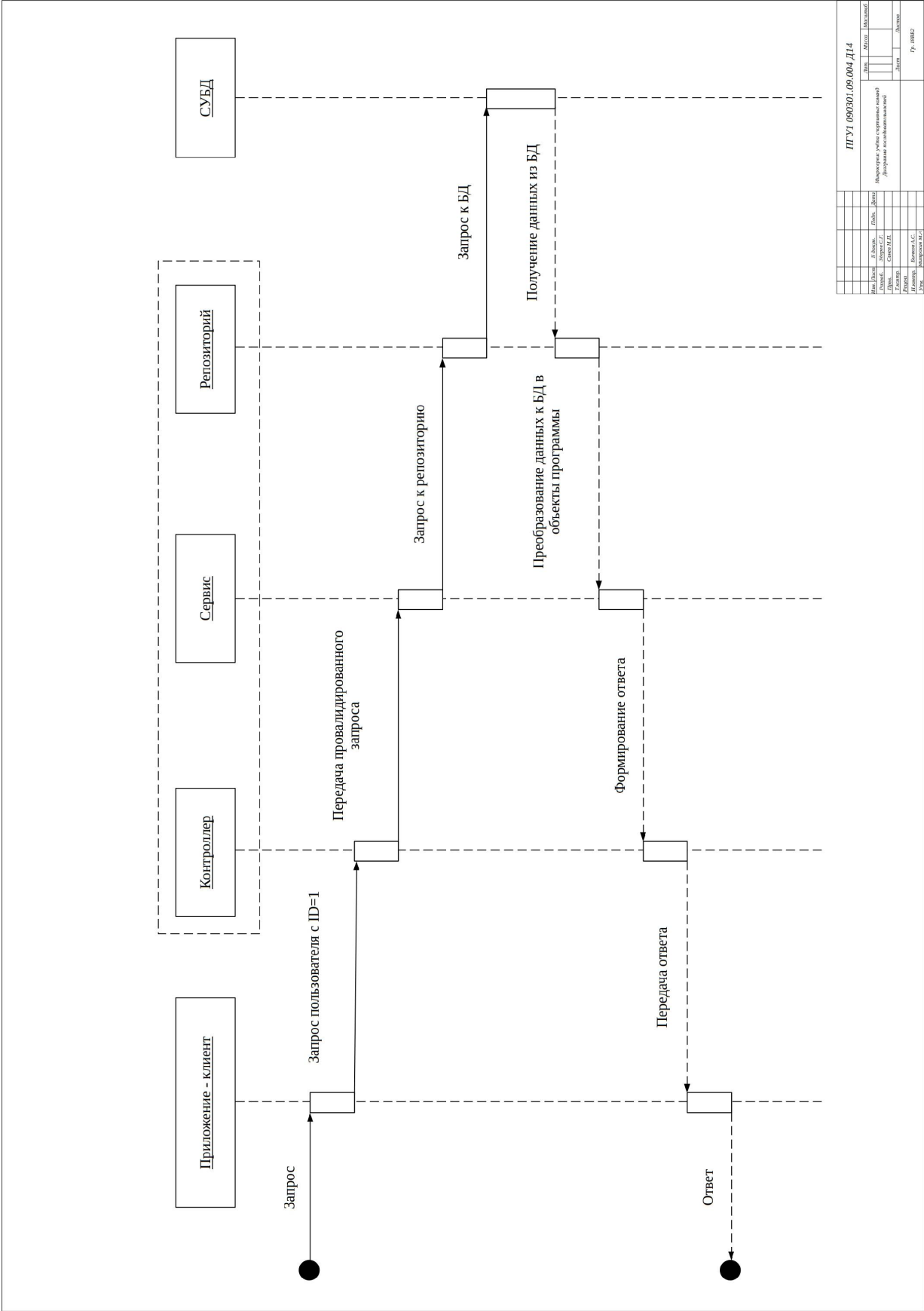
Лист

47

Приложение А.2 – Диаграмма классов



Приложение А.3 – Диаграмма последовательностей



ПГУ1 090301.09.004 Д14									
Имя	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Фамилия	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Должность	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Подпись	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Дата	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Страна	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Город	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Улица	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов
Почта	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов	И.И.	Иванов

Приложение Б

(обязательное)

Микросервис учета спортивных команд при проведении турниров

Модель базы данных

Приложение В

(обязательное)

Микросервис учета спортивных команд при проведении турниров

Описание *REST API*

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		52

Запрос «Создание вида спорта»

Метод: POST

URI: /sportType

Тело запроса:

```
{
  "name": "string"
}
```

Ответ:

```
201      Created
      {
        "id": "string",
        "name": "string"
      }
```

```
400      Bad Request
      { "errorMessage": "string" }
401      Not authorized
      { "errorMessage": "string" }
403      Forbidden
      { "errorMessage": "string" }
500      Server error
      { "errorMessage": "string" }
```

Запрос «Обновление вида спорта»

Метод: PUT

URI: /sportType/{id}

Тело запроса:

```
{
  "id": "string",
  "name": "string"
}
```

Ответ:

```
200      OK
      {
        "id": "string",
        "name": "string"
      }
```

```
400      Bad Request
      { "errorMessage": "string" }
401      Not Authorized
      { "errorMessage": "string" }
403      Forbidden
      { "errorMessage": "string" }
500      Server Error
      { "errorMessage": "string" }
```

Запрос «Удаление»

Метод: DELETE

URI: /{id}

Ответ:

204 No Content
400 Bad Request
 { "errorMessage": "string" }
401 Not Authorized
 { "errorMessage": "string" }
404 Not Found
 { "errorMessage": "string" }
500 Server Error
 { "errorMessage": "string" }

Запрос «Создание команды»

Метод: POST

URI: /team

Тело запроса:

```
{  
  "name": "string",  
  "sportTypeId": "string"  
}
```

Ответ:

201 Created
 {
 "id": "string",
 "name": "string",
 "sportType": {
 "id": "string",
 "name": "string"
 }
 }

400 Bad Request
 { "errorMessage": "string" }
401 Not authorized
 { "errorMessage": "string" }
403 Forbidden
 { "errorMessage": "string" }
500 Server error
 { "errorMessage": "string" }

Запрос «Обновление вида спорта»

Метод: PUT

URI: /sportType/{id}

Тело запроса:

```
{  
  "id": "string",  
  "name": "string",  
}
```

```

        "sportTypeId": "string"
    }

    Ответ:
    200      OK
    {
        "id": "string",
        "name": "string",
        "sportType": {
            "id": "string",
            "name": "string"
        },
    }

    400      Bad Request
    { "errorMessage": "string" }

    401      Not Authorized
    { "errorMessage": "string" }

    403      Forbidden
    { "errorMessage": "string" }

    500      Server Error
    { "errorMessage": "string" }

```

Запрос «Удаление команды»

```

Метод: DELETE
URI: /{id}
Ответ:
204      No Content
400      Bad Request
    { "errorMessage": "string" }
401      Not Authorized
    { "errorMessage": "string" }
404      Not Found
    { "errorMessage": "string" }
500      Server Error
    { "errorMessage": "string" }

```

Запрос «Создание игрока»

```

Метод: POST
URI: /player
Тело запроса:
    {
        "firstData": "string",
        "lastData": "string",
        "name": "string",
        "sportTypeId": "string",
        "teamId": "string"
    }

    Ответ:
    201      Created

```

```

{
  "firstData": "string",
  "id": "string",
  "lastData": "string",
  "name": "string",
  "sportType": {
    "id": "string",
    "name": "string"
  },
  "team": {
    "id": "string",
    "name": "string"
  },
}

```

```

400      Bad Request
        { "errorMessage": "string" }
401      Not authorized
        { "errorMessage": "string" }
403      Forbidden
        { "errorMessage": "string" }
500      Server error
        { "errorMessage": "string" }

```

Запрос «Обновление игрока»

Метод: PUT

URI: /player/{id}

Тело запроса:

```

{
  "firstData": "string",
  "lastData": "string",
  "name": "string",
  "sportTypeId": "string",
  "teamId": "string"
}

```

Ответ:

```

200      OK
        {
          "firstData": "string",
          "id": "string",
          "lastData": "string",
          "name": "string",
          "sportType": {
            "id": "string",
            "name": "string"
          },
          "team": {
            "id": "string",
            "name": "string"
          },
        }

```

```

    }
400      Bad Request
        { "errorMessage": "string" }
401      Not Authorized
        { "errorMessage": "string" }
403      Forbidden
        { "errorMessage": "string" }
500      Server Error
        { "errorMessage": "string" }

```

Запрос «Удаление игрока»

Метод: DELETE

URI: /{id}

Ответ:

```

204      No Content
400      Bad Request
        { "errorMessage": "string" }
401      Not Authorized
        { "errorMessage": "string" }
404      Not Found
        { "errorMessage": "string" }
500      Server Error
        { "errorMessage"/: "string" }

```

Приложение Г

(обязательное)

Микросервис учета спортивных команд при проведении турниров

Листинг программы

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		58

BaseRequest

```
package ru.zderev.sport.base;

import com.sun.istack.NotNull;

import java.io.Serializable;
import java.util.Objects;

public class BaseRequest implements Serializable {
    public static class Id implements Serializable {
        @NotNull
        private Long id;

        public Long getId() {
            return id;
        }

        public void setId(Long id) {
            this.id = id;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof Id)) return false;
            Id id1 = (Id) o;
            return Objects.equals(id, id1.id);
        }

        @Override
        public int hashCode() {
            return Objects.hash(id);
        }
    }
}
```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		59

```

public static class GenId<T> implements Serializable {
    @NotNull
    private T id;

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }
}

public static class GenValue<T> implements Serializable {
    @NotNull
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
}

```

MessageConfig

```

package ru.zderev.sport.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.context.support.ReloadableResourceBundleMess

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		60

```

ageSource;

@Configuration
public class MessageConfig {

    @Bean(name = "messageSource")
    public ReloadableResourceBundleMessageSource messageSource()
    {
        ReloadableResourceBundleMessageSource messageBundle =
new ReloadableResourceBundleMessageSource();
        messageBundle.setBasename("classpath:Messages");
        messageBundle.setDefaultEncoding("UTF-8");
        return messageBundle;
    }
}

```

HistoryController

```

package ru.zderev.sport.core.history.web;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import ru.zderev.sport.core.history.HistoryService;

@RestController
@RequestMapping("/history")
@Api(value = "PlayerController", tags = "Операции с историей")
public class HistoryController {
    final private HistoryService historyService;

    @Autowired
    public HistoryController(HistoryService historyService) {

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		61

```

        this.historyService = historyService;
    }

    @PostMapping
    @ApiOperation(value = "Получение записей")
    public void getAll() {
        historyService.findAll();
    }
}

```

History

```

package ru.zderev.sport.core.history;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;
import ru.zderev.sport.core.stortType.SportType;
import ru.zderev.sport.core.team.Team;

import javax.persistence.*;

@Data
@Entity
@Table(name = "app_history")
@NoArgsConstructor
public class History {

    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

    private String name;

    private String firstData;
}

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		62

```

        private String lastData;

        @ManyToOne
        @JoinColumn(name = "team")
        private Team team;

        @ManyToOne
        @JoinColumn(name = "sportyType")
        private SportType sportType;
    }

```

HistoryRepo

```

package ru.zderev.sport.core.history;

import org.springframework.data.jpa.repository.JpaRepository;

public interface HistoryRepo extends JpaRepository<History,
String> {
}

```

HistoryService

```

package ru.zderev.sport.core.history;

import org.springframework.stereotype.Service;
import ru.zderev.sport.core.player.Player;
import ru.zderev.sport.core.player.web.PlayerBaseReq;
import ru.zderev.sport.core.player.web.PlayerCreateReq;

import java.util.List;

@Service

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		63

```

public class HistoryService {

    final private HistoryRepo historyRepo;

    public HistoryService(HistoryRepo historyRepo) {
        this.historyRepo = historyRepo;
    }

    public void prepare(History history, PlayerBaseReq req){
        history.setName(req.getName());
        history.setFirstData(req.getFirstData());
        history.setLastData(req.getLastData());
        history.setSportType(req.getSportType());
        history.setTeam(req.getTeam());
    }

    public History create(PlayerCreateReq req) {
        History history = new History();
        this.prepare(history, req);
        return historyRepo.save(history);
    }

    public List<History> findAll() {
        return historyRepo.findAll();
    }

}

```

PlayerBaseReq

```

package ru.zderev.sport.core.player.web;

import io.swagger.annotations.ApiModelProperty;
import lombok.Data;
import ru.zderev.sport.base.BaseRequest;
import ru.zderev.sport.core.stortType.SportType;

```

```

import ru.zderev.sport.core.team.Team;

@Data
public class PlayerBaseReq extends BaseRequest {

    @ApiModelProperty(value = "Полное имя игрока")
    private String name;

    @ApiModelProperty(value = "Дата вступления в команду")
    private String firstData;

    @ApiModelProperty(value = "Дата окончания нахождения в
команде")
    private String lastData;

    @ApiModelProperty(value = "Вид спорта в котором выступает")
    private SportType sportType;

    @ApiModelProperty(value = "Команда за которую всутпает")
    private Team team;

}

```

PlayerController

```

package ru.zderev.sport.core.player.web;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.web.PageableDefault;
import org.springframework.web.bind.annotation.*;
import ru.zderev.sport.core.player.Player;
import ru.zderev.sport.core.player.PlayerService;

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		65

```

import java.util.List;

@RestController
@RequestMapping("/player")
@Api(value = "PlayerController", tags = "Операции с игроками")
public class PlayerController {
    private final PlayerService playerService;

    @Autowired
    public PlayerController(PlayerService playerService) {
        this.playerService = playerService;
    }

    @PostMapping
    @ApiOperation(value = "Создание записи об игроке")
    public void create(@RequestBody PlayerCreateReq req) {
        playerService.create(req);
    }

    @PutMapping("/{id}")
    @ResponseBody
    @ApiOperation(value = "Обновление записи об игроке")
    public Player update(@PathVariable String id,
                        @RequestBody PlayerBaseReq
req) {
        Player player = playerService.getOrThrow(id);
        return playerService.update(player, req);
    }

    @DeleteMapping("/{id}")
    @ApiOperation(value = "Удаление записи об игроке")
    public void delete(@PathVariable String id) {
        Player player = playerService.getOrThrow(id);
    }
}

```

```

        playerService.delete(player);
    }

    @GetMapping
    @ApiOperation(value = "Поиск по параметрам")
    public List<Player> findFilter(
        @RequestParam(required = false) String teamId,
        @RequestParam(required = false) String sportTypeId,
        @RequestParam(required = false) String firstData,
        @RequestParam(required = false) String lastData,
        @PageableDefault(sort = "id", direction =
Sort.Direction.ASC) Pageable pageable
    ) {
        if(teamId != null) {
            return playerService.findAllByTeamId(teamId);
        } else if(sportTypeId != null) {
            return
playerService.findAllBySportTypeId(sportTypeId);
        } else if(firstData != null || lastData != null) {
            return
playerService.findByBetweenFirstDataAndLastData(firstData,
lastData);
        }
        return playerService.findAll(pageable);
    }

    @GetMapping("/{id}")
    @ApiOperation(value = "Поиск записи об игроке по
идентификатору")
    public Player findById(@PathVariable String id) {
        return playerService.getOrThrow(id);
    }
}

```

PlayerCreateReq

```
package ru.zderev.sport.core.player.web;

import io.swagger.annotations.ApiModel;
import lombok.Data;

@Data
@ApiModel(value = "PlayerCreate", description = "Запрос на
создание сведений о игроке",
        parent = PlayerBaseReq.class)
public class PlayerCreateReq extends PlayerBaseReq {
}
```

PlayerUpdateReq

```
package ru.zderev.sport.core.player.web;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;
import lombok.Data;

@Data
@ApiModel(value = "PlayerUpdate", description = "Запрос на
изменение сведений о игроке",
        parent = PlayerBaseReq.class)
public class PlayerUpdateReq extends PlayerBaseReq {

    @ApiModelProperty(value = "Идентификатор игрока")
    private String id;

}
```

Player

```
package ru.zderev.sport.core.player;

import lombok.Data;
```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		68

```

import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;
import ru.zderev.sport.core.stortType.SportType;
import ru.zderev.sport.core.team.Team;

import javax.persistence.*;

@Data
@Entity
@Table(name = "app_player")
@NoArgsConstructor
public class Player {

    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

    private String name;

    private String firstData;

    private String lastData;

    @ManyToOne
    @JoinColumn(name = "team")
    private Team team;

    @ManyToOne
    @JoinColumn(name = "sportType")
    private SportType sportType;

}

```

PlayerRepo

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		69

```

package ru.zderev.sport.core.player;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface PlayerRepo extends JpaRepository<Player,
String> {

    List<Player> findAllByTeamId(String id);

    List<Player> findAllBySportTypeId(String id);

    List<Player> findByFirstDataAfterAndLastDataBefore(String
firstData, String lastData);

}

```

PlayerService

```

package ru.zderev.sport.core.player;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import ru.zderev.sport.core.player.web.PlayerBaseReq;
import ru.zderev.sport.core.player.web.PlayerCreateReq;
import ru.zderev.sport.core.sportType.SportType;
import ru.zderev.sport.core.sportType.SportTypeService;
import ru.zderev.sport.core.team.Team;
import ru.zderev.sport.core.team.TeamService;
import ru.zderev.sport.exception.player.PlayerIdEmptyException;
import ru.zderev.sport.exception.sport.SportTypeEmptyException;
import ru.zderev.sport.exception.team.TeamEmptyException;

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		70

```

import ru.zderev.sport.util.MessageUtil;
import ru.zderev.sport.exception.NotFoundException;

import javax.transaction.Transactional;
import java.util.List;

@Service
public class PlayerService {

    private final PlayerRepo playerRepo;
    private final SportTypeService sportTypeService;
    private final TeamService teamService;
    private final MessageUtil messageUtil;

    @Autowired
    public PlayerService(PlayerRepo playerRepo, SportTypeService
sportTypeService, TeamService teamService, MessageUtil
messageUtil) {
        this.playerRepo = playerRepo;
        this.sportTypeService = sportTypeService;
        this.teamService = teamService;
        this.messageUtil = messageUtil;
    }

    public void prepare(Player player, PlayerBaseReq req){
        player.setName(req.getName());
        player.setFirstData(req.getFirstData());
        player.setLastData(req.getLastData());
        player.setSportType(req.getSportType());
        player.setTeam(req.getTeam());
    }

    public Player create(PlayerCreateReq req) {
        Player player = new Player();

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		71

```

        this.prepare(player, req);
        return playerRepo.save(player);
    }

    public Player update(Player player, PlayerBaseReq req) {
        this.prepare(player, req);
        return playerRepo.save(player);
    }

    @Transactional
    public void delete(Player player) {
        playerRepo.delete(player);
    }

    public Player getOrThrow(String id) {
        return playerRepo.findById(id)
            .orElseThrow(() -> new NotFoundException(
messageUtil.getMessage("PlayerNotFound")));
    }

    public List<Player> findAll(Pageable pageable) {
        return playerRepo.findAll(pageable.getSort());
    }

    public List<Player> findAllByTeamId(String id) {
        if (id == null || teamService.getOrThrow(id) == null)
throw new PlayerIdEmptyException();
        Team team = teamService.getOrThrow(id);
        if (team == null) throw new TeamEmptyException();
        return playerRepo.findAllByTeamId(id);
    }

    public List<Player> findAllBySportTypeId(String id) {

```

```

        if (id == null || sportTypeService.getOrThrow(id) ==
null) throw new PlayerIdEmptyException();
        SportType sportType = sportTypeService.getOrThrow(id);
        if (sportType == null) throw new
SportTypeEmptyException();
        return playerRepo.findAllBySportTypeId(id);
    }

    public List<Player> findByBetweenFirstDataAndLastData(String
firstData, String lastData) {
        return
playerRepo.findByFirstDataAfterAndLastDataBefore(firstData,
lastData);
    }
}

```

SportTypeBaseReq

```

package ru.zderev.sport.core.stortType.web;

import io.swagger.annotations.ApiModelProperty;
import lombok.Data;
import ru.zderev.sport.base.BaseRequest;

@Data
public class SportTypeBaseReq extends BaseRequest {

    @ApiModelProperty(value = "Название спортивного типа")
    private String name;

}

```

SportTypeController

```

package ru.zderev.sport.core.stortType.web;

import io.swagger.annotations.Api;

```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		73

```

import io.swagger.annotations.ApiOperation;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.web.PageableDefault;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
import ru.zderev.sport.core.stortType.SportType;
import ru.zderev.sport.core.stortType.SportTypeService;
import ru.zderev.sport.core.team.TeamService;

import java.util.List;

@RestController
@RequestMapping(value = "/sport-type")
@Api(value = "SportTypeController", tags = "Операции с
спортивными типами")
public class SportTypeController {

    private final SportTypeService sportTypeService;
    private final TeamService teamService;

    public SportTypeController(SportTypeService
sportTypeService, TeamService teamService) {
        this.sportTypeService = sportTypeService;
        this.teamService = teamService;
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    @ResponseBody
    @ApiOperation(value = "Создание записи спортивного типа")
    public void create(@RequestBody SportTypeCreateReq req) {
        sportTypeService.create(req);
    }
}

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		74

```

@GetMapping
@ResponseBody
@ApiOperation(value = "Поиск всех записей спортивных типов")
public List<SportType> findAll(@PageableDefault(sort = "id",
direction = Sort.Direction.ASC) Pageable pageable) {
    return sportTypeService.findAll(pageable);
}

@GetMapping("/{id}")
@ResponseBody
@ApiOperation(value = "Поиск записи спортивного типа по
идентификатору")
public SportType getById(@PathVariable String id) {
    return sportTypeService.getOrThrow(id);
}

@PutMapping("/{id}")
@ResponseBody
@ApiOperation(value = "Обновление записи спортивного типа")
public SportType updateSportType(@PathVariable String id,
                                @RequestBody
SportTypeUpdateReq req) {
    SportType sportType = sportTypeService.getOrThrow(id);
    return sportTypeService.update(sportType, req);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
@ApiOperation(value = "Удаление записи спортивного типа")
public void delete(@PathVariable String id) {
    SportType sportType = sportTypeService.getOrThrow(id);
    sportTypeService.delete(sportType);
}

```

```
}
```

SportTypeCreateReq

```
package ru.zderev.sport.core.stortType.web;
```

```
import io.swagger.annotations.ApiModel;
```

```
import lombok.Data;
```

```
@Data
```

```
@ApiModel(value = "SportTypeCreateReq", description = "Запрос на  
создание общих сведений об виде спортивного типа",
```

```
parent = SportTypeBaseReq.class)
```

```
public class SportTypeCreateReq extends SportTypeBaseReq {
```

```
}
```

SportTypeUpdateReq

```
package ru.zderev.sport.core.stortType.web;
```

```
import io.swagger.annotations.ApiModel;
```

```
import io.swagger.annotations.ApiModelProperty;
```

```
import lombok.Data;
```

```
@ApiModel(value = "SportTypeUpdate", description = "Запрос на  
изменение сведений об виде спорта",
```

```
parent = SportTypeBaseReq.class)
```

```
@Data
```

```
public class SportTypeUpdateReq extends SportTypeBaseReq {
```

```
    @ApiModelProperty(value = "Идентификатор спортивного типа")
```

```
    private String id;
```

```
}
```

SportType

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		76

```

package ru.zderev.sport.core.stortType;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;
import ru.zderev.sport.core.player.Player;
import ru.zderev.sport.core.team.Team;

import javax.persistence.*;
import java.util.List;

@Data
@Entity
@Table(name = "app_sport_type")
@NoArgsConstructor
public class SportType {

    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

    private String name;

}

```

SportTypeRepo

```

package ru.zderev.sport.core.stortType;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface SportTypeRepo extends JpaRepository<SportType,
String> {

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		77

```

        SportType findByName(String name);
    }

```

SportTypeService

```

package ru.zderev.sport.core.stortType;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import ru.zderev.sport.core.stortType.web.SportTypeBaseReq;
import ru.zderev.sport.core.stortType.web.SportTypeCreateReq;
import ru.zderev.sport.core.stortType.web.SportTypeUpdateReq;
import ru.zderev.sport.util.MessageUtil;
import ru.zderev.sport.exception.NotFoundException;

import javax.transaction.Transactional;
import java.util.List;

@Service
public class SportTypeService {

    private final SportTypeRepo sportTypeRepo;
    private final MessageUtil messageUtil;

    @Autowired
    public SportTypeService(SportTypeRepo sportTypeRepo,
        MessageUtil messageUtil) {
        this.sportTypeRepo = sportTypeRepo;
        this.messageUtil = messageUtil;
    }

    public void create(SportTypeCreateReq req) {
        SportType sportType = new SportType();
        this.prepare(sportType, req);
        sportTypeRepo.save(sportType);
    }

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		78

```

    }

    public List<SportType> findAll(Pageable pageable) {
        return sportTypeRepo.findAll(pageable.getSort());
    }

    public SportType getOrThrow(String id) {
        return sportTypeRepo.findById(id).orElseThrow(() ->
            new
NotFoundException(messageUtil.getMessage("SportType.NotFound")))
;
    }

    public SportType update(SportType sport, SportTypeUpdateReq
req) {
        this.prepare(sport, req);
        return sportTypeRepo.save(sport);
    }

    @Transactional
    public void delete(SportType sportType) {
        sportTypeRepo.delete(sportType);
    }

    public void prepare(SportType sport, SportTypeBaseReq req){
        sport.setName(req.getName());
    }

}

```

TeamBaseReq

```

package ru.zderev.sport.core.team.web;

import io.swagger.annotations.ApiModelProperty;
import lombok.Data;

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		79

```

import ru.zderev.sport.base.BaseRequest;
import ru.zderev.sport.core.stortType.SportType;

@Data
public class TeamBaseReq extends BaseRequest {

    @ApiModelProperty(value = "Название команды")
    private String name;

    @ApiModelProperty(value = "Название спортивного типа
команды")
    private SportType sportType;

}

```

TeamController

```

package ru.zderev.sport.core.team.web;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.web.PageableDefault;
import org.springframework.web.bind.annotation.*;
import ru.zderev.sport.core.team.Team;
import ru.zderev.sport.core.team.TeamService;

import java.util.List;

@RestController
@RequestMapping("/team")
@Api(value = "SportTypeController", tags = "Операции с
командами")
public class TeamController {

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		80

```

private final TeamService teamService;

@Autowired
public TeamController(TeamService teamService) {
    this.teamService = teamService;
}

@PostMapping
@ApiOperation(value = "Создание записи о команде")
public void create(@RequestBody TeamCreateReq req) {
    teamService.create(req);
}

@PutMapping("/{id}")
@ResponseBody
@ApiOperation(value = "Обновление данных о команде")
public Team update(@PathVariable String id, @RequestBody
TeamUpdateReq req) {
    Team team = teamService.getOrThrow(id);
    return teamService.update(team, req);
}

@DeleteMapping("/{id}")
@ApiOperation(value = "Удаление команды")
public void deleteById(@PathVariable String id) {
    Team team = teamService.getOrThrow(id);
    teamService.delete(team);
}

@GetMapping
@ResponseBody
@ApiOperation(value = "Поиск команд по идентификатору

```

```

спортивной команды")

    public List<Team> findAllBySportTypeId(
        @RequestParam(required = false) String sportTypeId,
        @PageableDefault(sort = "id", direction =
Sort.Direction.ASC) Pageable pageable
    ) {
        if (sportTypeId == null) {
            return teamService.findAll(pageable);
        }
        return teamService.findAllBySportTypeId(sportTypeId);
    }

    @GetMapping("/{id}")
    @ApiOperation(value = "Поиск по идентификатору команды")
    public Team findById(@PathVariable String id) {
        return teamService.getOrThrow(id);
    }

}

```

TeamCreateReq

```

package ru.zderev.sport.core.team.web;

import io.swagger.annotations.ApiModel;
import lombok.Data;

@Data
@ApiModel(value = "TeamCreate", description = "Запрос на
создание сведений о команде",
        parent = TeamBaseReq.class)
public class TeamCreateReq extends TeamBaseReq {
}

```

TeamUpdateReq

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		82

```

package ru.zderev.sport.core.team.web;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;
import lombok.Data;

@Data
@ApiModel(value = "TeamUpdate", description = "Запрос на
изменение сведений об виде спорта",
        parent = TeamBaseReq.class)
public class TeamUpdateReq extends TeamBaseReq
{

    @ApiModelProperty(value = "Идентификатор команды")
    private Long id;

}

```

Team

```

package ru.zderev.sport.core.team;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;
import ru.zderev.sport.core.player.Player;
import ru.zderev.sport.core.stortType.SportType;

import javax.persistence.*;
import java.util.List;

@Data
@Entity
@Table(name = "app_team")
@NoArgsConstructor
public class Team {

```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		83

```

@Id
@GeneratedValue(generator = "uuid")
@GenericGenerator(name = "uuid", strategy = "uuid2")
private String id;

private String name;

@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "sportType")
private SportType sportType;
}

```

TeamRepo

```

package ru.zderev.sport.core.team;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import ru.zderev.sport.core.team.Team;
import ru.zderev.sport.core.team.web.TeamUpdateReq;

import java.util.List;

@Repository
public interface TeamRepo extends JpaRepository<Team, String> {

    List<Team> findAllBySportTypeId(String id);

}

```

TeamService

```

package ru.zderev.sport.core.team;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Pageable;

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		84

```

import org.springframework.stereotype.Service;
import ru.zderev.sport.core.stortType.SportTypeRepo;
import ru.zderev.sport.core.stortType.SportTypeService;
import ru.zderev.sport.core.team.web.TeamBaseReq;
import ru.zderev.sport.core.team.web.TeamCreateReq;
import ru.zderev.sport.exception.team.TeamIdEmptyException;
import ru.zderev.sport.util.MessageUtil;
import ru.zderev.sport.exception.NotFoundException;

import javax.transaction.Transactional;
import java.util.List;

@Service
public class TeamService {

    private final TeamRepo teamRepo;
    private final SportTypeRepo sportTypeRepo;
    private final SportTypeService sportTypeService;
    private final MessageUtil messageUtil;

    @Autowired
    public TeamService(TeamRepo teamRepo, SportTypeRepo
sportTypeRepo, SportTypeService sportTypeService, MessageUtil
messageUtil) {
        this.teamRepo = teamRepo;
        this.sportTypeRepo = sportTypeRepo;
        this.sportTypeService = sportTypeService;
        this.messageUtil = messageUtil;
    }

    public void prepare(Team team, TeamBaseReq req){
        team.setName(req.getName());
        team.setSportType(req.getSportType());
    }

```

```

public void create(TeamCreateReq req) {
    Team team = new Team();
    sportTypeService.getOrThrow(req.getSportType().getId());
    this.prepare(team, req);
    teamRepo.save(team);
}

public Team update(Team team, TeamBaseReq req) {
    this.prepare(team, req);
    return teamRepo.save(team);
}

@Transactional
public void delete(Team team) {
    teamRepo.delete(team);
}

public Team getOrThrow(String id) {
    return teamRepo.findById(id)
        .orElseThrow(() -> new NotFoundException(
messageUtil.getMessage("TeamNotFound")));
}

public List<Team> findAll(Pageable pageable) {
    return teamRepo.findAll(pageable.getSort());
}

public List<Team> findAllBySportTypeId(String id) {
    if (id == null || sportTypeService.getOrThrow(id) ==
null) throw new TeamIdEmptyException();
    sportTypeService.getOrThrow(id);
    return teamRepo.findAllBySportTypeId(id);
}

```

```

    }

}

```

MessageUtil

```

package ru.zderev.sport.util;

import org.springframework.context.MessageSource;
import org.springframework.context.i18n.LocaleContextHolder;
import org.springframework.stereotype.Service;

import java.util.Locale;

@Service
public class MessageUtil {
    private final MessageSource messageSource;

    public MessageUtil(final MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public String getMessage(String selector, Object...
parameters) {
        return messageSource.getMessage(selector, parameters,
LocaleContextHolder.getLocale());
    }

    public String getMessage(String selector) {
        return messageSource.getMessage(selector, new Object[]
{}, LocaleContextHolder.getLocale());
    }

    public String getMessage(String selector, Locale locale) {
        return messageSource.getMessage(selector, new Object[]
{}, locale);
    }
}

```

```

    }

    public String getMessage(String selector, Locale locale,
Object... parameters) {
        return messageSource.getMessage(selector, parameters,
locale);
    }
}

```

PlayerIdEmptyException

```

package ru.zderev.sport.exception.player;

import ru.zderev.sport.exception.AbstractException;
import ru.zderev.sport.util.MessageUtil;

import javax.persistence.EntityNotFoundException;

public class PlayerIdEmptyException extends AbstractException {

    private final MessageUtil messageUtil;

    public PlayerIdEmptyException() {
        super("Player id is empty!");
        throw new EntityNotFoundException("Player.NotFound");
    }

}

```

SportTypeEmptyException

```

package ru.zderev.sport.exception.sport;

import ru.zderev.sport.exception.AbstractException;
import javax.persistence.EntityNotFoundException;

public class SportTypeEmptyException extends AbstractException {

    public SportTypeEmptyException() {

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		88

```

        super("Sport type id is empty!");
        throw new EntityNotFoundException("SportType.NotFound");
    }
}

```

TeamEmptyException

```

package ru.zderev.sport.exception.team;

import ru.zderev.sport.exception.AbstractException;

import javax.persistence.EntityNotFoundException;

public class TeamEmptyException extends AbstractException {
    public TeamEmptyException() {
        super("Team is empty!");
        throw new EntityNotFoundException("Team.NotFound");
    }
}

```

TeamIdEmptyException

```

package ru.zderev.sport.exception.team;

import ru.zderev.sport.exception.AbstractException;

import javax.persistence.EntityNotFoundException;

public class TeamIdEmptyException extends AbstractException {
    public TeamIdEmptyException() {
        super("Sport type id is empty!");
        throw new EntityNotFoundException("Team.NotFound");
    }
}

```

AbstractException

```

package ru.zderev.sport.exception;

```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		89

```

public class AbstractException extends RuntimeException {

    protected AbstractException() {
    }

    protected AbstractException(Throwable cause) {
        super(cause);
    }

    protected AbstractException(String message) {
        super(message);
    }

}

```

ApplicationException

```

package ru.zderev.sport.exception;

public class ApplicationException extends RuntimeException {
    public ApplicationException() {
        super();
    }

    public ApplicationException(String message) {
        super(message);
    }

    public ApplicationException(String message, Throwable cause)
    {
        super(message, cause);
    }

}

```

NotFoundException

```

package ru.zderev.sport.exception;

```

					<i>ПГУ1 090301.09.001 ПЗ</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		90

```

public class NotFoundException extends ApplicationException {
    public NotFoundException() {
        super();
    }

    public NotFoundException(String message) {
        super(message);
    }

    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

MessageUtil

```

package ru.zderev.sport.util;

import org.springframework.context.MessageSource;
import org.springframework.context.i18n.LocaleContextHolder;
import org.springframework.stereotype.Service;

import java.util.Locale;

@Service
public class MessageUtil {
    private final MessageSource messageSource;

    public MessageUtil(final MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public String getMessage(String selector, Object...
parameters) {
        return messageSource.getMessage(selector, parameters,
LocaleContextHolder.getLocale());
    }
}

```

```

    }

    public String getMessage(String selector) {
        return messageSource.getMessage(selector, new Object[]
        {}, LocaleContextHolder.getLocale());
    }

    public String getMessage(String selector, Locale locale) {
        return messageSource.getMessage(selector, new Object[]
        {}, locale);
    }

    public String getMessage(String selector, Locale locale,
    Object... parameters) {
        return messageSource.getMessage(selector, parameters,
        locale);
    }
}

```

SportApplication

```

package ru.zderev.sport;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import
springfox.documentation.swagger2.annotations.EnableSwagger2;

```

```

@SpringBootApplication
@Configuration
@EnableSwagger2
public class SportApplication {

    public static void main(String[] args) {
        SpringApplication.run(SportApplication.class, args);
    }

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }

}

```

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		93

Приложение Д

(обязательное)

Микросервис учета загрузки работников предприятия

Презентация

					ПГУ1 090301.09.001 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись	Дата		94

Выпускная квалификационная работа
бакалавра на тему «Микросервис учёта
спортивных команд при проведении турниров»

Подготовил: Здерев С.Г., группа 18ВВ2
Руководитель: к.т.н., доцент каф. ВТ Синев М.П.

Проблема – информационная поддержка детско-юношеского спорта

Решение – учёт детско-юношеских команд в различных видах спорта.

Возможности подобных приложений:

- ▶ удобство проведения спортивных мероприятий в школах;
- ▶ Отслеживание успехов и прогресса молодого поколения, что в будущем позволяет более крупным организаторам собирать команды;
- ▶ автоматизация большинства организационных моментов для клиента;
- ▶ наглядное отражает развития детско-юношеского спорта в регионах.

Анализ аналогов

ВХЛ

- ▶ разделение на клубы и игроков;
- ▶ обширный спектр фильтрации списка игроков по всем параметрам и алфавиту;
- ▶ широкий спектр информации о каждом клубе;
- ▶ понятный для пользователя интерфейс.

R-Hockey

- ▶ широкий спектр лиг, начиная с юношеской;
- ▶ состав команд на выбранный временной период с статистикой игроков по ролям

Стек технологий

- ▶ язык программирования *JAVA*
- ▶ *Spring Framework*
- ▶ РСУБД *PostgreSQL*
- ▶ *IntelliJ IDEA*
- ▶ *Swagger*



PostgreSQL



Разделение архитектуры на функциональные блоки

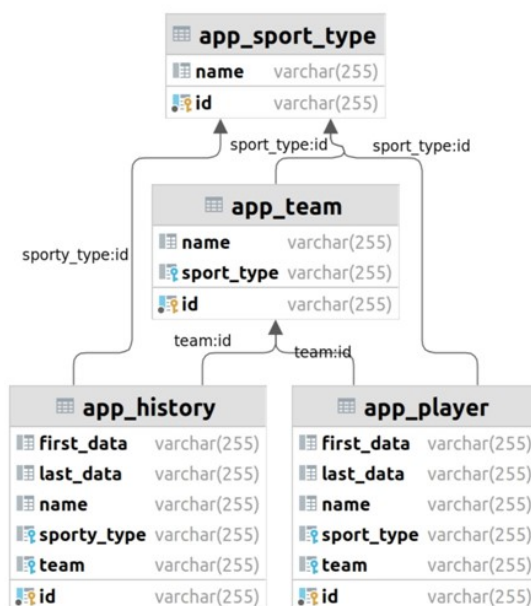
Основываясь на описанных ранее СУБД и *REST API*, для удобства взаимодействия архитектура разбита на следующие функциональные блоки:

- ▶ классы-сервисы, обеспечивают взаимодействие контроллеров с репозиториями, вмещают в себя общую массу бизнеслогики;
- ▶ классы-репозитории, отвечают за взаимодействие СУБД с сущностями, передаются из классов-сервисов соответственно;
- ▶ классы-сущности, описывают сущности БД в программном коде, помогают во взаимодействии с данными, полученными из классов-репозитория;
- ▶ классы-контроллеры, выполняющие запросы *REST API*, валидируют полученные данные, возвращают ответ.

Диаграмма прецедентов



Физическая модель базы данных



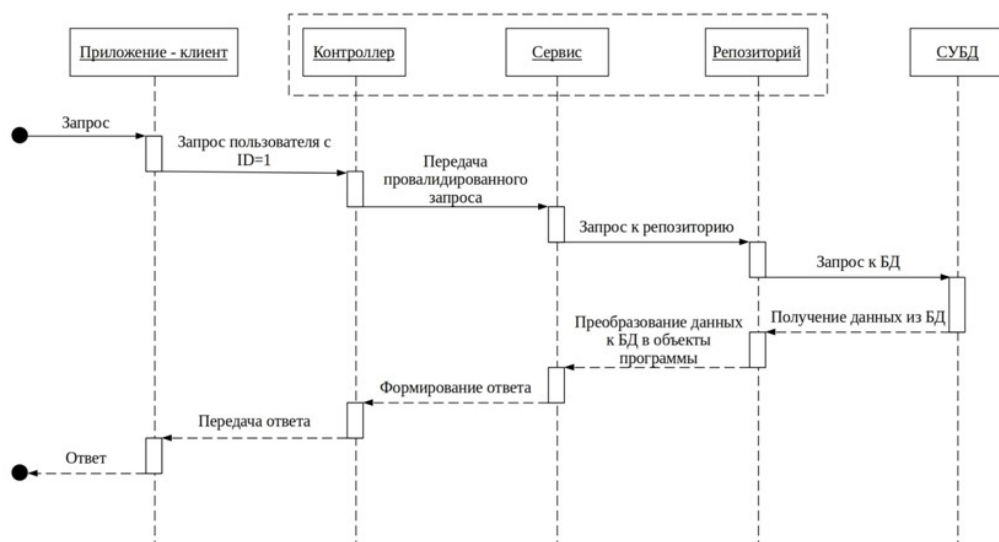
Основной функционал

- ▶ создание вида спорта
- ▶ обновление данных вида спорта
- ▶ удаление вида спорта
- ▶ поиск видов спорта
- ▶ создание команды
- ▶ обновление данных команды
- ▶ удаление команды
- ▶ поиск команд
- ▶ создание игрока
- ▶ обновление данных игрока
- ▶ удаление игрока
- ▶ поиск игрока
- ▶ поиск игрока по определённой дате вступления/выхода в команду
- ▶ поиск по диапазону дат нахождения в команде
- ▶ поиск игрока по команде
- ▶ поиск игрока по виду спорта

Функциональные блоки микросервиса

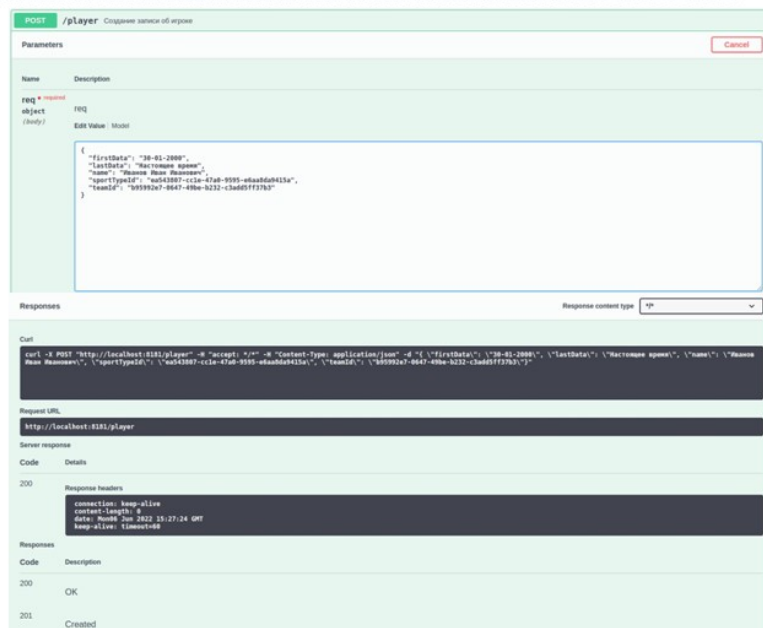
- ▶ контроллеры
- ▶ сущности
- ▶ сервисы
- ▶ репозитории
- ▶ представления

Диаграмма последовательностей



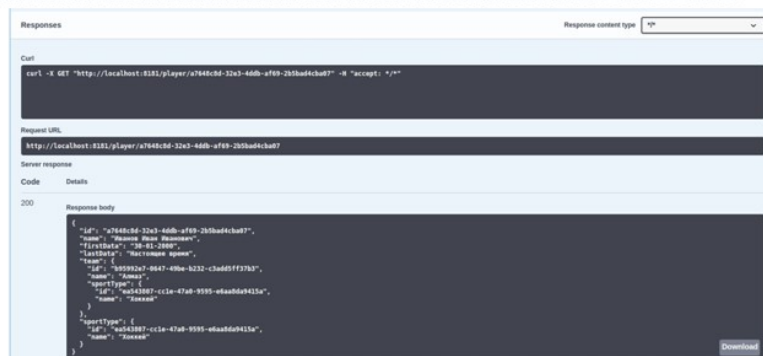
Результат выполнения запроса на создание игрока

«POST /player»



Результат выполнения запроса на получение игроков

«GET /player/{id}»



Заключение

В ходе выполнения данной работы была изучена методология проектирования приложения с микросервисной архитектурой и приобретены навыки работы с фрейворком *Spring*. При реализации поставленной задачи были получены следующие результаты:

- ▶ разработана структура базы данных для СУБД *PostgreSQL*
- ▶ разработан прикладной интерфейс для взаимодействия с остальными микросервисами приложения (*REST API*)
- ▶ разработано и отлажено приложение «Микросервис учёта спортивных команд для проведения турниров».

Сервис поддерживает полный функционал для интеграции с остальными микросервисами спортивного ресурса.

