

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

Программные средства поддержки работы
спортивной организации

Студент

(подпись, дата)

Лыгин Константин Андреевич
(ФИО полностью)

Руководитель

(подпись, дата)

Самуйлов С.В.
(фамилия, инициалы)

Нормоконтролёр

(подпись, дата)

Такташкин Д.В.
(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026г. № 12)

Заведующий кафедрой

(подпись)

Козлов А.Ю.
(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

(подпись)

Попова Н.А.
(фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
А.Ю. Козлов
«26» декабря 2025 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Лыгин Константин Андреевич гр. 22ВП1 факультета ВТ

<Фамилия, имя, отчество полностью>

2. Тема работы Программные средства поддержки работы спортивной организации

Тема утверждена приказом ПГУ № 1583/о от "30" декабря 2025 г.

3. Руководитель работы к.т.н., доцент кафедры «МО и ПЭВМ» Самуйлов С.В.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Разработать информационную систему управления спортивной организацией.

2) Среда функционирования: web-браузер под управлением ОС Windows 10 и выше.

3) Среда разработки: Windows 10/11.

Языковые средства разработки: C#, TypeScript, SQL, UML.

Инструментальные средства: JetBrains Rider, JetBrains DataGrip, .NET 9, ASP.NET Core, React Native, PostgreSQL, Entity Framework Core, ClickHouse, MinIO, Docker Compose, Swagger, xUnit, Testcontainers, PlantUML, GitHub.

Технологии разработки: клиент-серверная архитектура, REST API, объектно-ориентированное программирование, модульная архитектура, CQRS.

4) Основные функции приложения:

а) регистрация и авторизация пользователей;

б) просмотр филиалов спортивной организации;

в) просмотр услуг, помещений, оборудования и расписания филиала;

г) оформление абонемента клиентом;

д) запись клиента на тренировочное занятие;

е) управление профилем пользователя;

ж) создание и управление филиалами;

- з) управление помещениями, оборудованием и изображениями;
 и) управление услугами, расписанием, занятиями и сотрудниками;
 к) создание и удаление шаблонов абонементов;
 л) просмотр аналитики по активности клиентов и филиалов.

5. Перечень подлежащих разработке вопросов _____

- 1) Анализ предметной области;
 2) Проектирование программных средств;
 3) Реализация программных средств;
 4) Тестирование программных средств;
 5) Оформление пояснительной записки.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	07.04.2026 – 13.04.2026	
2	Проектирование программных средств	20%	14.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	27.04.2026 – 11.05.2026	
4	Тестирование программных средств	20%	12.05.2026 – 21.05.2026	
5	Оформление пояснительной записки	20%	22.05.2026 – 02.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы _____ 26.12.2025 г.
 (подпись, дата)

Задание к исполнению принял студент _____ 26.12.2025 г.
 (подпись, дата)

Работу к защите допустить
 Декан ФВТ _____ Л.Р. Фионова
 (подпись, дата)

Реферат

Пояснительная записка содержит 149 листов, 35 рисунка, 18 таблиц, 39 использованных источников, 6 приложений.

КЛИЕНТСКАЯ ЧАСТЬ, СИСТЕМА КОНТРОЛЯ ОТПУСКОВ, ВЕБ-ПРИЛОЖЕНИЕ, REACT NATIVE, TYPESCRIPT, POSTGRESQL, CLICKHOUSE, MINIO, JWT, КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА, UML, DOCKER, REST API

Объектом разработки является информационная система управления спортивной организацией.

Цель работы – разработка клиент-серверной системы для автоматизации работы с филиалами, клиентами, услугами, расписанием, абонементом и записями на занятия.

Серверная часть реализована на C# с использованием ASP.NET Core. Для хранения данных применяется PostgreSQL, для изображений – MinIO, для аналитических событий – ClickHouse. Клиентская часть реализована на React Native с использованием TypeScript. Приложение позволяет клиентам просматривать филиалы, оформлять абонементы и записываться на занятия, а администраторам – управлять филиалами, помещениями, оборудованием, услугами, расписанием и сотрудниками.

Разработанная система может использоваться как основа для внедрения в спортивной организации или фитнес-центре.

					ПГУ 09.03.04 – 08КР221.12			
Изм.	Лист	№ докум.	Подп.	Дата	Программные средства поддержки работы спортивной организации. Пояснительная записка	Лит.	Лист	Листов
Разраб.		Лыгин К.А.						
Пров.		Самуйлов С.В.					4	149
						Группа 22ВП1		
Н. контр.		Такташкин Д.В.						
Утверд.		Козлов А.Ю.						

Содержание

Введение.....	7
1 Постановка задачи и анализ требований	8
1.1 Постановка задачи на разработку приложения	8
1.2 Анализ программ-аналогов.....	8
1.2.1 Анализ системы Gymdesk	8
1.2.2 Анализ системы PushPress	10
1.2.3 Анализ системы GymMaster	11
1.2.4 Сравнение систем управления спортивными организациями	12
1.3 Анализ функциональных требований.....	13
1.4 Анализ нефункциональных требований.....	17
2 Проектирование приложения.....	21
2.1 Архитектура приложения.....	21
2.2 Проектирование структур данных	23
2.2.1 Концептуальная модель базы данных	23
2.2.2 Логическая модель базы данных.....	25
2.3 Проектирование серверной части	27
3 Разработка программного обеспечения	29
3.1 Разработка физической модели базы данных	29
3.2 Реализация серверной части информационной системы.....	34
3.3 Реализация клиентской части информационной системы.....	40
3.4 Реализация интерфейса приложения	45
4 Отладка и тестирование программного обеспечения	62
4.1 Выбор режима тестирования	62
4.2 Модульное тестирование	63
4.3 Интеграционное тестирование	65
4.4 Функциональное тестирование.....	67
5 Контроль качества программного обеспечения.....	71
5.1 Выбор метрик программного кода.....	71
5.2 Оценка качества программного кода	72

5.2.1 Метрика SLOC	72
5.2.2 Метрика Холстеда.....	73
5.2.3 Метрика Чепина.....	75
5.2.4 Метрика Р. Мартина	76
5.2.5 Индикатор Maintainability Index	77
6 Планирование разработки и оценка бюджета	79
Заключение	82
Список использованных источников	85
Приложение А (справочное) Глоссарий	89
Приложение Б (обязательное) Серверная часть приложения	95
Приложение В (обязательное) Клиентская часть приложения	103
Приложение Г (рекомендуемое) Листинг модульных тестов	111
Приложение Д (рекомендуемое) Листинг интеграционных тестов	121
Приложение Е (рекомендуемое) Графическая часть	129

Введение

Актуальность выбранной темы связана с тем, что в современных условиях спортивные клубы и фитнес-центры работают с большим количеством данных: учет клиентов, расписание занятий, запись на услуги, управление филиалами и персоналом. При отсутствии единой цифровой системы эти процессы часто выполняются разрозненно, что усложняет работу сотрудников, повышает вероятность ошибок и снижает качество обслуживания клиентов [1, 2].

Целью работы является проектирование и реализация программной системы, которая позволит автоматизировать основные бизнес-процессы спортивной организации и обеспечить удобное взаимодействие пользователей с сервисом. Для достижения поставленной цели необходимо решить ряд задач: проанализировать предметную область, определить функциональные и нефункциональные требования, спроектировать архитектуру, реализовать серверную и клиентскую части, а также провести проверку работоспособности разработанного решения [3].

Объектом исследования является процесс управления деятельностью спортивной организации, а предметом исследования – методы и средства разработки клиент-серверных информационных систем для автоматизации процесса.

Практическая значимость работы заключается в том, что разрабатываемая система может быть использована как основа для внедрения в реальной организации с возможностью дальнейшего расширения функциональности. В процессе выполнения работы особое внимание уделяется не только реализации пользовательских сценариев, но и вопросам надежности, безопасности, производительности и сопровождаемости программного продукта.

1 Постановка задачи и анализ требований

1.1 Постановка задачи на разработку приложения

На начальном этапе выполнения дипломной работы формулируется задача разработки информационной системы управления спортивной организацией, предназначенной для цифровизации ключевых процессов обслуживания клиентов и администрирования филиалов.

В ходе анализа требований были выделены две основные группы пользователей: клиентов и сотрудников. Для клиентов система должна быть простой и понятной, пользователь должен быстро войти в систему, найти нужную информацию и выполнить целевое действие, например записаться на услугу. Для управляющей части важны структурированное управление данными, контроль прав доступа и удобная работа с объектами системы. Таким образом, уже на старте проектирования становится очевидно, что система должна учитывать разные сценарии использования и обеспечивать корректную работу каждой роли.

Также при анализе данных были учтены нефункциональные требования [4], так как именно они определяют качество будущего программного продукта, исходя из того, что система должна работать стабильно, обеспечивать приемлемое время отклика, защищать данные пользователей и сохранять возможность дальнейшего расширения. Дополнительно важно, чтобы решение было удобным в сопровождении, с понятной архитектурой, логичной структурой модулей и возможностью тестирования ключевых компонентов.

1.2 Анализ программ-аналогов

1.2.1 Анализ системы Gymdesk

Gymdesk – это облачное программное решение для управления фитнес-клубами, студиями единоборств и других клубов с членством [5]. Платформа призвана автоматизировать большую часть административной работы: от

регистрации и оплаты абонементов до расписания занятий и маркетинга. Интерфейс представлен на рисунке 1.

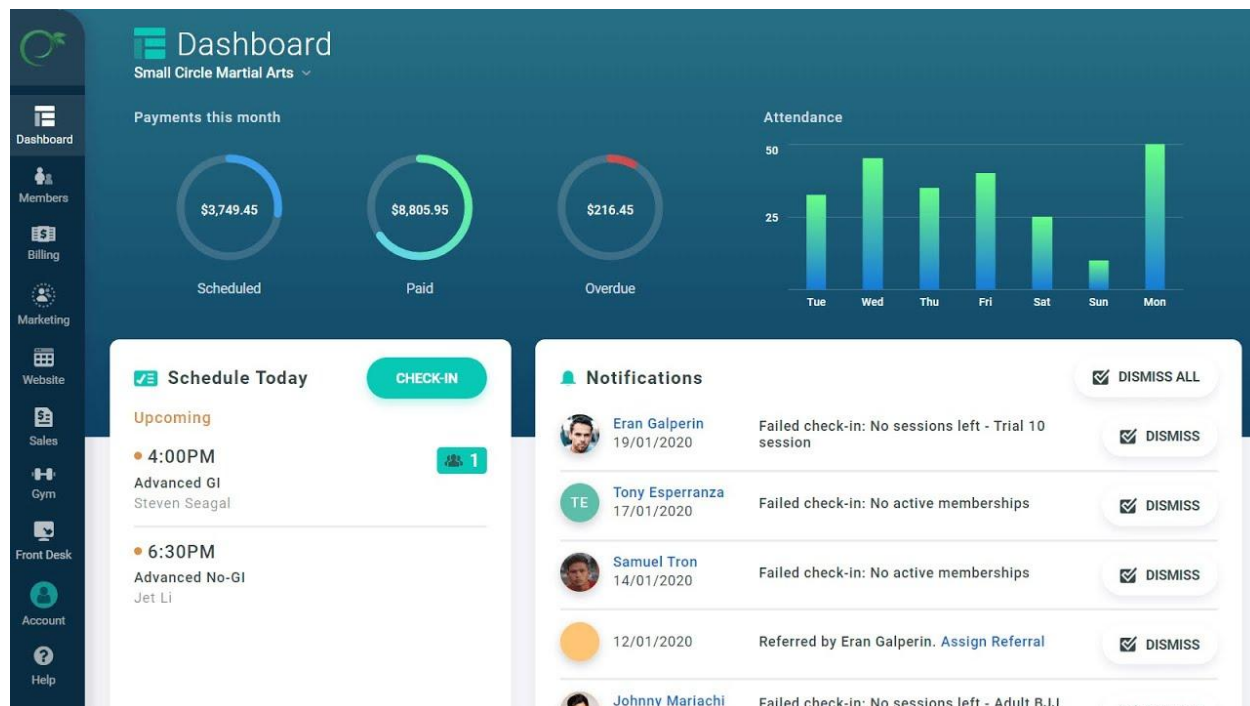


Рисунок 1 – Интерфейс продукта «Gymdesk»

Среди сильных сторон можно отметить полное управление членством, Gymdesk позволяет создавать различные планы абонементов, обрабатывать рекуррентные платежи и управлять статусами клиентов. Автоматизированная биллинговая система, интеграция с такими платежными провайдерами, как Stripe, Square и др., позволяет автоматически выставлять счета, уведомлять о просроченных оплатах, и снижать администраторскую нагрузку. Расписание классов и посещаемость, есть календарь для расписания, инструкторы могут задавать свое расписание, а пользователи записываются и получают напоминания.

Тем не менее, данная система обладает определенными недостатками. В частности, функционал мобильного приложения ограничен по сравнению с десктопной версией. Пользователи указывают на недостаточную глубину возможностей при создании пользовательских отчетов.

Gymdesk будет отличным вариантом для средних и небольших фитнес-клубов, студий единоборства, йоги и подобных бизнесов, которые хотят автоматизировать членство, расписание, оплату и коммуникацию без сложных настроек.

1.2.2 Анализ системы PushPress

PushPress – платформа, разработанная специально для владельцев залов, студий и академий [6]. Она упрощает повседневные операции: расписания классов, платежи, продажи и аналитику. Интерфейс представлен на рисунке 2.

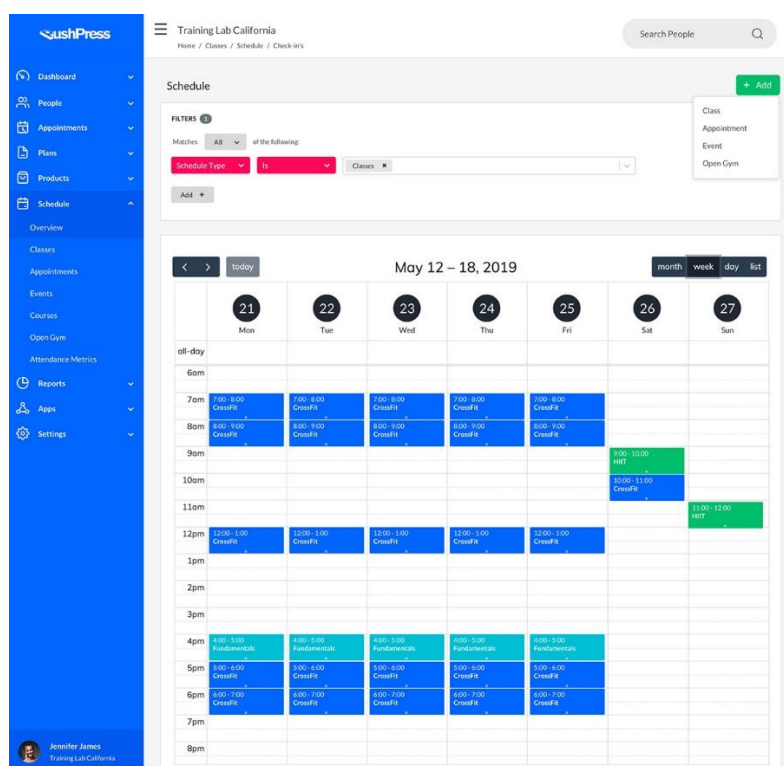


Рисунок 2 – Интерфейс продукта «PushPress»

Сильными сторонами данной платформы являются, бронирование и класс-планирование тренировок, клиенты могут резервировать классы, а тренеры и менеджеры – настраивать расписание, управлять списками ожиданиями и отменами. Есть мобильное приложение для клиентов, чтобы записаться, смотреть свое расписание и оплачивать. Интеграция с Google Календарь для напоминаний.

PushPress хороший выбор для небольших и средних залов, CrossFit-студий, студий групповых занятий, которым нужно простое, но мощное решение для управления клиентами, классами и платежами, без чрезмерной сложности.

1.2.3 Анализ системы GymMaster

GymMaster – облачное программное обеспечение для управления фитнес-клубами, разработанное для автоматизации многих админ-процессов: управление абонементом, расписаниями, платежами и коммуникацией. Интерфейс представлен на рисунке 3.

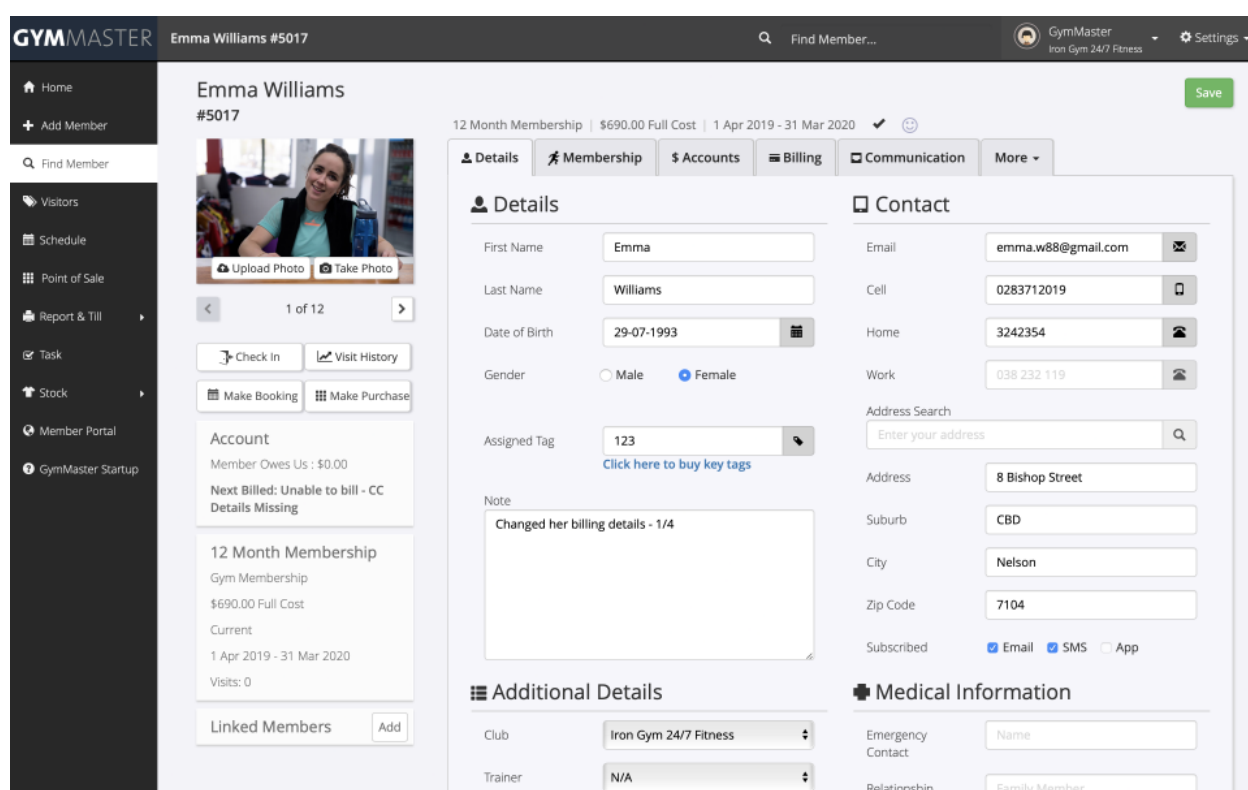


Рисунок 3 – Интерфейс продукта «GymMaster»

Можно выделить сильные стороны данного ПО, рассылки для email и SMS, уведомление клиентов. Контроль доступа, позволяет настроить системы доступа (например, через карты, штрихкоды) для контроля посещения. Представляет детальные отчеты о посещаемости, доходах, активности клиентов. GymMaster позволяет автоматизировать регистрацию, продление абонементов, отмены и управление профилем клиентов [7].

Из минусов можно выделить несколько моментов. Из-за богатства функционала может быть сложнее в освоении, особенно для небольших клубов с ограниченными IT-ресурсами. Также из-за широкого набора функций и интеграций цена может быть выше, особенно для маленьких студий.

GymMaster – хорош для фитнес-клубов среднего и крупного размера, которые нуждаются в широком функционале: расписания, контроля доступа, автоматизации финансов, отчетности и маркетинга.

1.2.4 Сравнение систем управления спортивными организациями

Для оценки и сравнения информационных систем, используемых в системах управления спортивными организациями, был разработан перечень критериев. На их основе проводится анализ имеющихся решений (таблица 1). Оценки даны по 0 – 2 баллам для каждой функции:

- 0 баллов: нет функции вообще;
- 1 балл: присутствует базовая функция, ограниченная;
- 2 балла: функция развита, глубокая, имеет широкие возможности.

Таблица 1 – Оценки функциональных возможностей систем

Критерии	Gymdesk	PushPress	GymMaster
Расписание и запись	2	1	2
Мобильное приложение	2	2	2
Отслеживание процессов	1	1	2
Отчеты и аналитика	1	1	2
Управление абонементом	2	2	2
Работоспособность в России	0	0	0
Итого	8	7	10

Для более детального описания ключевых функций наиболее значимых сценариев применения были подготовлены их спецификации.

Спецификация варианта использования «Авторизоваться» подробнее рассматривается в таблице 2.

Таблица 2 – Спецификация варианта использования «Авторизоваться»

Прецедент: Авторизоваться
ID: 1
Краткое описание: Пользователь авторизуется в системе и получает доступ согласно роли
Главные актеры: Пользователь
Второстепенные актеры: Нет
Предусловие: <ol style="list-style-type: none"> 1. Пользователь не авторизован 2. Открыта форма входа
Основной поток: <ol style="list-style-type: none"> 1. Пользователь вводит логин и пароль 2. Нажимает кнопку «Войти» 3. Система проверяет учетные данные 4. Система выдает токен доступа
Постусловия: <ol style="list-style-type: none"> 1. Пользователь авторизован 2. Токен сохранен в сессии

Спецификация варианта использования «Зарегистрироваться» подробнее рассматривается в таблице 3.

Таблица 3 – Спецификация варианта использования «Зарегистрироваться»

Прецедент: Зарегистрироваться
ID: 2
Краткое описание: Пользователь создает новую учетную запись
Главные актеры: Пользователь
Второстепенные актеры: Нет

Продолжение таблицы 3

Предусловие: <ol style="list-style-type: none"> 1. Пользователь не авторизован 2. Открыта форма регистрации
Основной поток: <ol style="list-style-type: none"> 1. Пользователь вводит регистрационные данные 2. Нажимает кнопку «Зарегистрироваться» 3. Система валидирует данные и проверяет уникальность 4. Система создает аккаунт пользователя
Постусловия: <ol style="list-style-type: none"> 1. Аккаунт пользователя создан 2. Пользователь может выполнить вход

Спецификация варианта использования «Оформить абонемент» подробнее рассматривается в таблице 4.

Таблица 4 – Спецификация варианта использования «Оформить абонемент»

Прецедент: Оформить абонемент
ID: 3
Краткое описание: Клиент оформляет абонемент на выбранную услугу
Главные актеры: Клиент
Второстепенные актеры: Нет
Предусловие: <ol style="list-style-type: none"> 1. Клиент авторизован 2. Услуга с ценой существует в системе
Основной поток: <ol style="list-style-type: none"> 1. Клиент выбирает услуги и параметры абонемента 2. Подтверждает оформление 3. Система проверяет корректность параметров 4. Система создает абонемент
Постусловия: <ol style="list-style-type: none"> 1. Абонемент создан 2. Абонемент доступен для просмотра

Спецификация варианта использования «Создать тренировочную сессию» подробнее рассматривается в таблице 5.

Таблица 5 – Спецификация варианта использования «Создать тренировочную сессию»

Прецедент: Создать тренировочную сессию
ID: 4
Краткое описание: Управляющий создает тренировочную сессию
Главные актеры: Управляющий
Второстепенные актеры: Нет
Предусловие: <ol style="list-style-type: none"> 1. Управляющий авторизован 2. Для услуги существует расписание
Основной поток: <ol style="list-style-type: none"> 1. Управляющий открывает раздел тренировочных сессий 2. Выбирает услугу, тренера, зал и время 3. Подтверждает создание 4. Система проверяет корректность и отсутствие конфликтов 5. Система сохраняет тренировочную сессию
Постусловия: <ol style="list-style-type: none"> 1. Тренировочная сессия создана 2. Сессия доступна для просмотра и дальнейшей записи клиентами

Спецификация варианта использования «Создать филиал» подробнее рассматривается в таблице 6.

Таблица 6 – Спецификация варианта использования «Создать филиал»

Прецедент: Создать филиал
ID: 5
Краткое описание: Управляющий создает новый филиал спортивной организации
Главные актеры: Управляющий
Второстепенные актеры: Нет
Предусловие: Управляющий авторизован

Продолжение таблицы 6

Основной поток:
1. Управляющий открывает раздел филиалов
2. Вводит данные филиала
3. Подтверждает создание
4. Система валидирует и сохраняет данные
Постусловия:
1. Филиал создан
2. Филиал доступен для просмотра клиентам

Анализ функциональных требований приводит к созданию вариантов использования приложения. Они демонстрируют каждый этап взаимодействия пользователей с системой. Добавление спецификаций к этим вариантам поможет лучше спроектировать систему.

1.4 Анализ нефункциональных требований

В рамках работы мной разрабатывается система управления спортивной организацией, ориентированная на использования в филиальной структуре с ролями клиента, сотрудника и управляющего. При постановке задачи я исхожу из того, что система должна обеспечивать не только функциональную полноту, но и устойчивое качество работы в реальных условиях эксплуатации. Поэтому на этапе проектирования закладываются нефункциональные требования, определяющие ожидаемый уровень производительности, надежности, безопасности, масштабируемости, сопровождаемости и удобства использования.

Проект будет представлять собой клиент-серверную информационную систему для управления спортивной организацией. Клиентская часть будет реализована на React Native с поддержкой Android, IOS и Web, а серверная часть будет построена на ASP.NET Core 9 с PostgreSQL, ClickHouse, MinIO, Hangfire и развертывание через Docker Compose с проксированием через nginx. Такая архитектура позволяет разделить пользовательский интерфейс, прикладную бизнес-логику, основное реляционное хранилище, объектное хранилище изображений, фоновые задачи и аналитическое решение.

С точки зрения производительности разрабатываемая система должна обеспечивать быстрый отклик пользовательского интерфейса и API при типовой и повышенной нагрузке. Для основных операций чтения данных, включая просмотр филиалов, помещений, расписания и услуг, время ответа должно оставаться в пределах комфортного для пользователя интервала, а операции записи, такие как регистрация, авторизация и оформления абонемента, должны стабильно выполняться и без ошибок. Архитектурно это будет выполняться разделением клиентской и серверной частей, применением эффективного взаимодействия с базой данных и оптимизацией сетевого обмена через единый API-шлюз.

Отдельным требованием предъявляется к работе аналитическими данными. Система должна фиксировать пользовательские и бизнес-события в структурном виде, чтобы управляющий мог получать сводку по филиалам и анализировать динамику активности. Аналитический модуль должен поддерживать построение трендов по дням, часам и десятиминутным интервалам, а также фильтрацию по филиалу и выбранной дате. Такой подход повышает наблюдаемость системы и позволяет принимать управленческие решения на основе накопленных данных.

Требования надежности предполагают, что система будет корректно запускаться и функционировать в контейнерной среде, а также сохранять данные при перезапусках и обновлении компонентов. Для этого в целевой реализации предусматривается использование устойчивого хранилища данных, автоматизированного старта инфраструктурных сервисов и контролируемой инициализации внешних зависимостей. Дополнительно система должна поддерживать регламент резервного копирования и сценарий восстановления, что обеспечит предсказуемое восстановление работоспособности при сбоях.

Важным нефункциональным требованием является корректная работа с часовыми поясами. Система используется в контексте расписания, тренировочных сессий и записей клиентов, поэтому некорректная интерпретация даты и времени может привести к ошибкам обслуживания. Дополнительно предусмотрена конвертация между IANA- и Windows-

идентификаторами, что повышает переносимость приложения между разными операционными системами.

Безопасность является обязательным свойством проектируемого решения. Доступ к защищенным операциям будет осуществляться только после аутентификации и проверки прав пользователя в соответствии с его ролью. Конфиденциальные данные, включая учетные записи и персональную информацию, должны обрабатываться с соблюдением принципов минимально необходимого доступа. В проекте будет использоваться токенная модель аутентификации, безопасное хранение паролей в виде криптографических хэшей, защита транспортного уровня посредством HTTPS и разграничение доступа к управляющим интерфейсам. Управление секретами будет организовано вне исходного кода, через параметры окружения и защищенные конфигурационные механизмы.

Масштабируемость системы закладывается как возможность дальнейшего роста без архитектурного пересмотра основных модулей. Серверная часть проектируется таким образом, чтобы поддерживать расширение за счет увеличения числа экземпляров API и развития инфраструктуры хранения и фоновой обработки задач. Такой подход позволит адаптировать систему к увеличению числа пользователей, филиалов и бизнес-операций при сохранении целевых показателей производительности.

Сопровождаемость и расширяемость рассматриваются как ключевые требования для жизненного цикла программного продукта после завершения дипломного проекта. Программная архитектура будет построена модульно, с четким разделением ответственности между слоями, чтобы упростить развитие функциональности и внесения изменений. Для подтверждения корректности реализации предусматривается автоматизированное тестирование, включающее модульные и интеграционные сценарии.

С точки зрения удобства использования разрабатываемая система должна представлять понятный пользовательский поток для ключевых действий: регистрации, входа, просмотра информации о филиалах и услугах, оформления

записи и управления профильными данными. Интерфейс должен быть последовательным, предсказуемым и поддерживать информативную обратную связь при ошибках ввода и сетевых исключениях. Для управляющего сегмента важно обеспечить логическую организацию операций управления объектами системы и расписанием.

2 Проектирование приложения

2.1 Архитектура приложения

При проектировании информационной системы управления спортивными организациями была выбрана клиент-серверная архитектура [8]. Данный подход позволяет разделить пользовательский интерфейс и серверную обработку данных, за счет чего достигается удобство сопровождения, возможность независимого развертывания компонентов и повышение устойчивости системы при росте нагрузки.

Разрабатываемое программное решение включает клиентское приложение SportManagementClient и серверное приложение SportManagementSystem. Клиентская часть обеспечивает взаимодействие пользователя с системой, формирование и отправку запросов, отображение полученных данных и выполнение пользовательских сценариев. Серверная часть принимает запросы, выполняет проверку прав доступа, реализует бизнес-логику, взаимодействует с хранилищами данных и возвращает результат в стандартизированном формате.

В архитектурном отношении серверная часть реализована как многослойное монолитное приложение с модульной организацией предметной области [9, 10, 11]. В системе выделены модули Users, Assets, Services, Clients, Scheduling и Images. Для каждого модуля применена единая структура, включающая доменный слой, слой прикладной логики, инфраструктурный слой и слой контрактов обмена данными. Такое построение обеспечивает четкое разделение ответственности между частями приложения и позволяет локализовать изменения внутри соответствующего модуля.

На уровне API используются контроллеры ASP.NET Core с маршрутизацией вида «v1/[controller]». Обработка прикладных сценариев выполняется через MediatR по модели команд и запросов CQRS [12], что позволяет сделать контроллеры тонкими и перенести основную логику в обработчики. Валидация входных данных реализована средствами

FluentValidation и выполняется в конвейере обработки запроса, благодаря чему обеспечивается единый подход к проверке корректности данных.

Слой хранения данных построен на базе Entity Framework Core и СУБД PostgreSQL. В классе «ApplicationDbContext» определены основные сущности предметной области, а для критичных полей настроены ограничения уникальности, что поддерживает целостность данных. Для хранения файлов изображений используется объектное хранилище MinIO, при этом в реляционной базе сохраняются только метаданные и идентификаторы файлов [13]. Такое разделение снижает нагрузку на основную базу и упрощает работу с медиа-контентом.

В системе также реализована фоновая обработка задач с использованием Hangfire. Хранение служебных данных планировщика выполняется в PostgreSQL [14].

Клиентская часть реализована на React Native [15] с поддержкой Web, Android и iOS. Внутренняя структура организована по feature-ориентированному принципу: отдельно выделены прикладные экраны, функциональные модули, доменные сущности и общие переиспользуемые компоненты. Для обращения к серверу используется централизованный HTTP-клиент, который формирует запросы к API, добавляет JWT-токен в заголовок авторизации для защищенных методов и приводит ошибки к единому виду для корректной обработки на уровне интерфейса.

Развертывание системы выполняется в контейнерной среде Docker Compose [16, 17], где отдельно поднимаются сервисы приложения, базы данных, обратного прокси и объектного хранилища. Такой подход обеспечивает воспроизводимость окружения и упрощает запуск решения на разных рабочих станциях.

2.2 Проектирование структур данных

2.2.1 Концептуальная модель базы данных

При проектировании базы данных для моей системы управления спортивной организацией я исходил из того, что модель должна отражать реальные бизнес-объекты и их естественные связи [18, 19]. В центре модели находится сущность учетной записи пользователя `UserAccount`, которая отвечает за доступ в систему и хранит данные аутентификации: электронную почту, хэш пароля, роль и статус. С учетной записью логически связаны профили двух типов пользователей: `Client` и `Staff`. Такой подход позволяет разделить предметные данные пользователя, при этом сохранить единый механизм авторизации.

Организационная структура представлена сущностью `Branch` (филиал). С филиалом связаны помещения `Room`, а с помещениями – оборудование `Equipment`, что формирует иерархию инфраструктуры спортивной организации. Для визуального представления объектов используется сущность `Image`: изображения могут быть привязаны к филиалам, помещениям и оборудованию, а также использоваться как аватары клиента и сотрудника. Это делает модель универсальной для работы с медиафайлами без дублирования логики хранения.

Блок услуг реализован через сущность `SportService`, где каждая услуга имеет уникальный код, название, категорию и признак активности. Для корректного учета стоимости введена сущность `ServicePrice`, которая хранит цену услуги, валюту и период действия цены. Благодаря этому можно фиксировать изменения стоимости во времени, не теряя историю тарифов. На основе услуг строится расписание `ServiceSchedule`, где задаются день недели, временной интервал, сотрудник и помещение, в котором оказывается услуга.

Фактическое проведение занятий отражается сущностью `TrainingSession`, которая может быть создана на базе расписания и содержит конкретные дату, время, тренера и статус занятия. Запись клиента на занятие хранится в сущности `Booking`, связывающей `Client` и `TrainingSession` с фиксацией даты записи и текущего статуса. Отдельно выделена сущность `Membership`, представляющая

абонемент клиента на конкретную услугу: в ней хранится период действия, общее и оставшееся число посещений, а также статус абонемента.

На рисунке 5 представлена полная концептуальная модель базы данных в графическом виде.

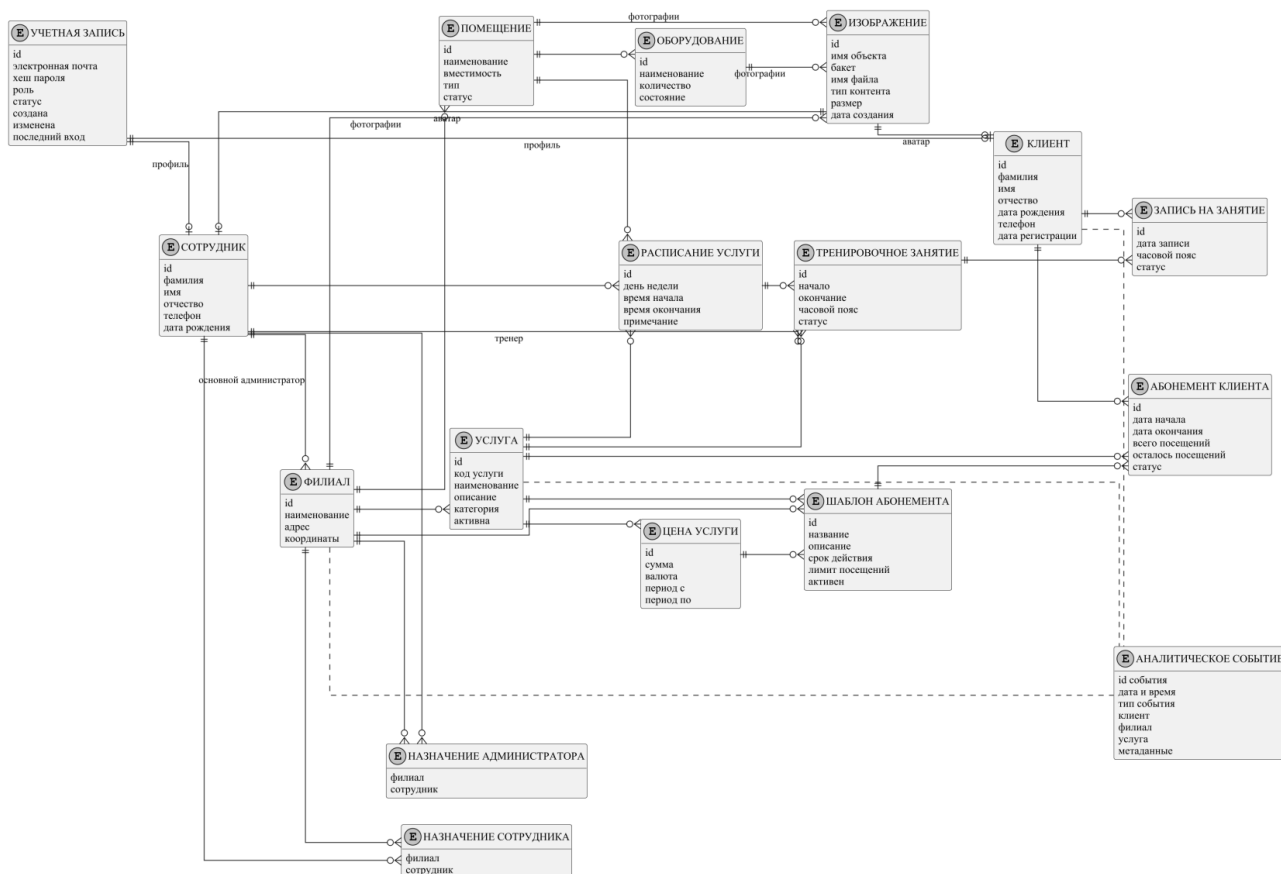


Рисунок 5 – Концептуальная модель базы данных

С точки зрения целостности данных концептуальная модель опирается на первичные и внешние ключи, а также на бизнес-ограничения, включая уникальность электронной почты у учетной записи и уникальность кода услуги. Для доменных состояний используется перечисления (роли, статусы аккаунтов, записи, абонементов, тренировок, состояния оборудования и помещений), что делает данные согласованными и уменьшает вероятность некорректных значений. В результате сформированная концептуальная модель охватывает все ключевые процессы проекта и обеспечивает основу для дальнейшего логического и физического проектирования базы данных.

2.2.2 Логическая модель базы данных

На этапе логического проектирования базы данных я преобразую концептуальную модель предметной области в набор взаимосвязанных таблиц с четко определенными атрибутами, ключами и ограничениями целостности. В моем проекте логическая модель ориентирована на реляционный подход и реализуется в СУБД PostgreSQL. Основная цель логического уровня заключается в том, чтобы формально зафиксировать структуру хранения данных и обеспечить корректную поддержку всех пользовательских и административных сценариев системы управления спортивными организациями.

На рисунке 6 представлена логическая модель базы данных в графическом виде.

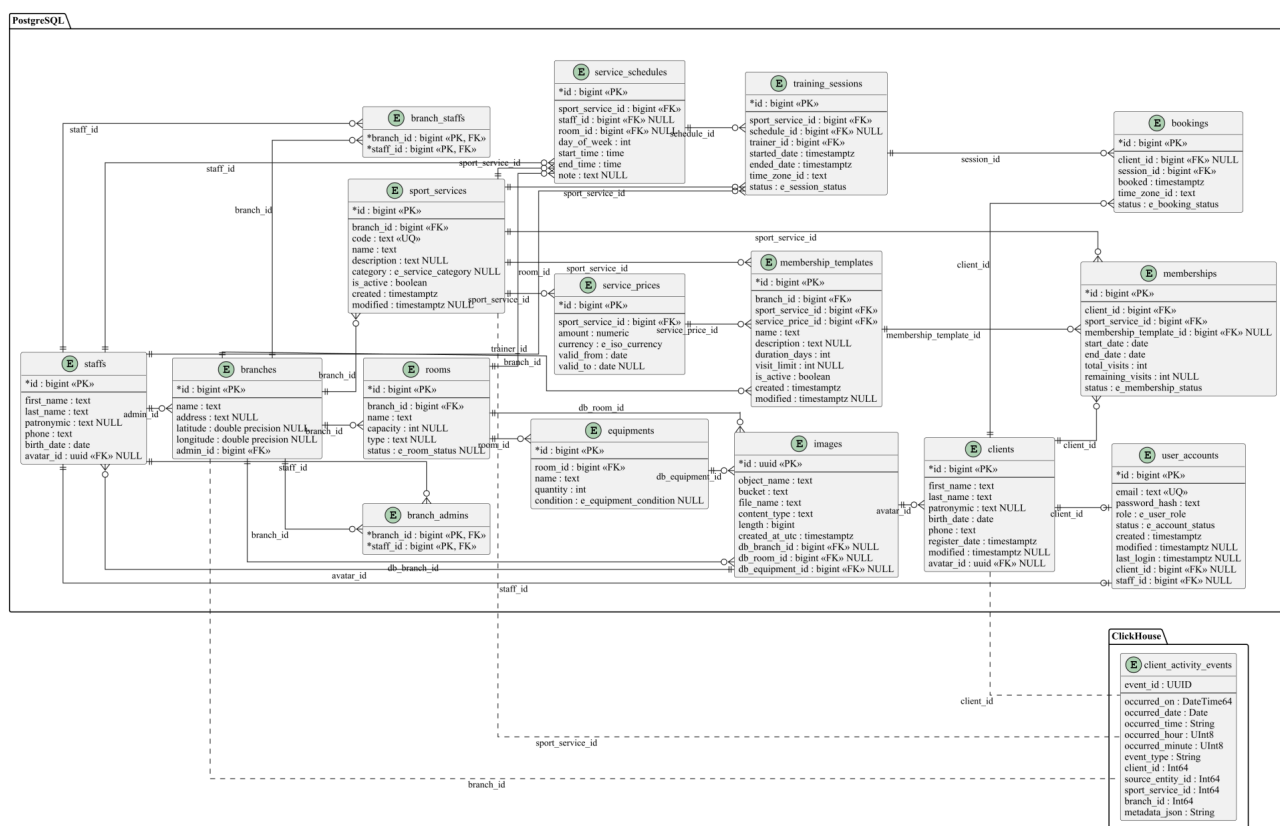


Рисунок 6 – Логическая модель базы данных

Логическая модель включает несколько взаимосвязанных модулей. Модуль пользователей представлен таблицами user_accounts, clients и staffs. Таблица user_accounts хранит учетные данные для входа в систему, роли и

статусы учетных записей. Таблицы `clients` и `staffs` содержат профильные данные клиентов и сотрудников соответственно. Связь учетной записи с конкретным профилем задается через внешние ключи, что позволяет использовать единую схему аутентификации и авторизации при сохранении разделения предметных данных.

Модуль инфраструктуры спортивной организации реализован таблицами `branches`, `rooms` и `equipment`. Таблица `branches` описывает филиалы, `rooms` хранит информацию о помещениях конкретного филиала, а `equipments` содержит перечень оборудования, закрепленного за помещениями. Такая структура обеспечивает иерархическое представление материальной базы организации и поддерживает дальнейшее расширение по числу филиалов и ресурсов.

Модуль услуг и расписания включает таблицы `sport_services`, `service_prices`, `service_schedules` и `training_session`. Таблица `sport_service` хранит справочник услуг с уникальным кодом и категорией. Таблица `service_prices` фиксирует стоимость услуги с периодом действия, что позволяет учитывать изменение тарифов во времени. Таблица `service_schedules` отражает шаблонное расписание, а `training_session` – фактические сессии (занятия), проводимые по расписанию и с участием конкретного сотрудника.

Модуль клиентских операций представлен таблицами `memberships` и `bookings`. В `memberships` хранится информация об абонентах клиентов на услуги с периодом действия и учетом количества посещений. В `bookings` фиксируется запись клиента на конкретную тренировочную сессию с указанием даты записи и текущего статуса. В совокупности эти таблицы обеспечивают поддержку основных бизнес-процессов взаимодействия клиента с организацией.

Для работы с медиафайлами в логической модели предусмотрена таблица `images`, которая хранит метаданные файлов в объектном хранилище. Эта таблица используется как источник ссылок на изображения филиалов, помещений, оборудования, а также на аватары клиентов и сотрудников.

С точки зрения ограничений целостности в модели используются первичные ключи для однозначной идентификации строк, внешние ключи для

поддержания ссылочной целостности, а также уникальные ограничения на уровне бизнес-атрибутов. Для статусов и ролей используются перечислимые доменные значения, что исключает сохранения некорректных состояний.

При формировании логической диаграммы я придерживался принципов нормализации данных, чтобы минимизировать избыточность и исключить аномалии вставки, обновления и удаления. Разделение сущностей на самостоятельные таблицы позволило сделать структуру базы данных понятной, расширяемой и удобной для дальнейшего физического проектирования и реализации в рамках серверной части работы.

2.3 Проектирование серверной части

Северная часть системы проектируется как самостоятельный прикладной слой, отвечающий за обработку бизнес-логики, хранения данных, проверку прав доступа и интеграцию с внешними сервисами. Основной задачей северной части является предоставление единого API для клиентских приложений (Android, iOS и Web), а также обеспечение надежной и безопасной обработки пользовательских запросов.

В качестве технологической платформы я использую ASP.NET Core. Такой выбор обусловлен поддержкой современного подхода к построению веб-сервисов, удобной интеграцией с реляционной базой данных, встроенными механизмами аутентификации и возможностью масштабирования. Архитектурно серверная часть проектируется модульно, функциональность разделяется на предметные модули, каждый из которых содержит контракты, прикладную логику, доменные сущности и инфраструктурные реализации. Это позволяет изолировать ответственность, упростить тестирование и сделать дальнейшее развитие системы более предсказуемым.

В основе обработки бизнес-операций лежит паттерн CQRS с использованием MediatR. Команды используются для изменения состояния системы, а запросы – для получения данных. Такой подход, делает код более читаемым и снижает связанность между контроллерами и внутренней логики.

Для проверки входных данных на сервере применяется FluentValidation, что позволяет централизованно задавать правила валидации и возвращать пользователю корректные сообщения об ошибках.

Для доступа к данным проектируется реляционная модель на PostgreSQL с использованием Entity Framework Core [20]. Применение миграции позволяет управлять изменениями схемы базы данных контролируемо и воспроизводимо в разных средах развертывания.

Серверная часть включает подсистему аутентификации и авторизации на основе JWT. После успешного входа пользователь получает токен доступа, который используется при обращении к защищенным маршрутам API. Дополнительно в проектировании учитывается разграничение прав на уровне ролей, чтобы управляющие функции были доступны только соответствующим категориям пользователей.

Для хранения файлов используется объектное хранилище MinIO. В базе данных хранится не сам бинарный контент, а метаданные файла и идентификатор объекта в хранилище. Такое решение уменьшает нагрузку на реляционную БД и повышает удобство дальнейшей работы с файлами.

В проекте серверной части также закладывается фоновая обработка задач с помощью Hangfire. Этот механизм используется для операций, которые должны выполняться по расписанию или асинхронно, без блокировки пользовательского запроса. Наличие фоновых задач делает поведение системы более устойчивым при росте числа операций.

Для обеспечения единообразной обработки ошибок проектируется централизованный middleware-слой, который перехватывает исключения и формирует стандартизованный ответ API. Такой подход упрощает клиентскую обработку ошибок и улучшает диагностируемость системы в процессе эксплуатации.

3 Разработка программного обеспечения

3.1 Разработка физической модели базы данных

При реализации информационной системы управления спортивной организацией особое внимание мной было уделено разработке базы данных, так как именно она обеспечивает хранение основных сведений о пользователях, филиалах услугах, расписаниях, тренировочных занятиях, абонементов и записях клиентов. База данных должна не только сохранять информацию, но и поддерживать целостность связей между объектами предметной области, поэтому при ее проектировании был выбран реляционный подход.

В качестве основной системы управления базами данных в проекте используется PostgreSQL. Данный выбор обусловлен поддержкой надежных транзакций, внешних ключей, индексов, перечислимых типов, а также удобной интеграцией с серверной частью, разработанной на ASP.NET Core. Взаимодействие приложения с базой данных реализовано с помощью Entity Framework Core и провайдером Npgsql [21]. Такой подход позволяет описывать структуру таблиц на уровне доменных классов C#, а затем автоматически формировать и применять миграции для изменения физической схемы базы данных.

Центральным элементом доступа к данным является класс ApplicationDbContext. В нем определены наборы данных DbSet, соответствующие основным таблицам системы: UserAccounts, Clients, Staffs, Branches, Rooms, Equipments, SportServices, ServicesPrices, ServiceSchedules, TrainingSession, Bookings, Memberships, MembershipsTemplates и Images. Кроме того в контексте настроены ограничения уникальности, связи между сущностями и перечислимые типы PostgreSQL, которые используются для хранения ролей пользователя, статусов аккаунта, состояний занятия, абонементов, бронирования, помещений и оборудования.

Физическая модель базы данных, сформированная на основе фактической схемы хранения PostgreSQL и аналитической таблицы ClickHouse, представлена на рисунке 7.

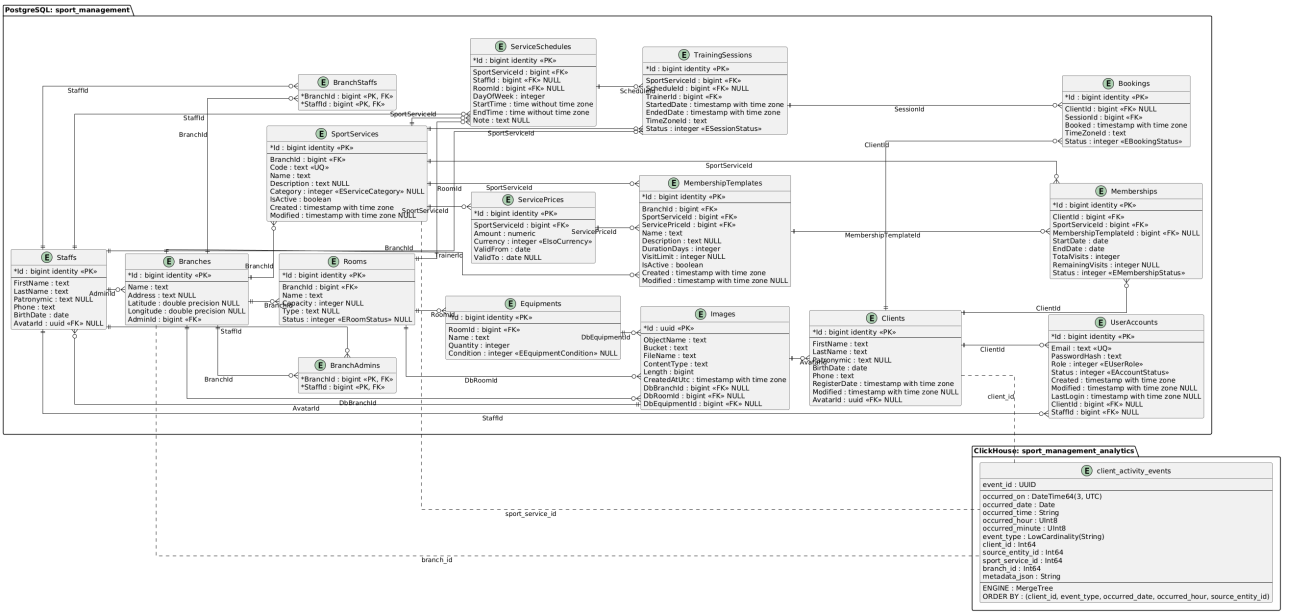


Рисунок 7 – Физическая модель базы данных информационной системы управления спортивными организациями

Основные таблицы базы данных приведены в таблице 7.

Таблица 7 – Основные таблицы базы данных

Таблица	Назначение
UserAccounts	Хранение учетных данных пользователя, роли, статуса, хэша пароля и времени последнего входа
Clients	Хранение профильных данных клиентов спортивных организаций
Staffs	Хранение данных сотрудников, тренеров и управляющих
Branches	Описание филиалов организации, адреса, координаты и управляющего филиала
Rooms	Хранение сведений о помещениях, относящихся к конкретному филиалу

Продолжение таблицы 7

Таблица	Назначение
Equipments	Хранение информации об оборудовании, закрепленном за помещением
SportServices	Справочник спортивных услуг с названием, категорией и признаком активности
ServicePrices	Хранение цен на услуги с указанием валюты и периода действия
ServiceSchedules	Описание регулярного расписания услуг по дням недели, времени, сотруднику и помещению
TrainingSessions	Хранение конкретных тренировочных сессий с датой, временем, тренером и статусом
Booking	Фиксация записей клиентов на тренировочные сессии
MembershipsTemplates	Хранение шаблонов абонементов, связанных с филиалом, услугой и ценой
Membership	Хранение приобретенных клиентами абонементов и количества доступных посещений
Images	Хранение метаданных изображений, размещенных в объектном хранилище

Структура базы данных разделена на несколько логических блоков. Первый блок связан с пользователями. Таблица UserAccounts содержит данные необходимые для входа в систему: электронную почту, хэш пароля, роль и статус учетной записи. Для исключения регистрации одной и той же почты на поле Email настроено уникальное ограничение. Профильные данные вынесены в отдельные таблицы Clients и Staffs. Благодаря этому учетные данные не смешиваются с предметной информацией, а механизм авторизации остается единым для клиентов и сотрудников.

Второй блок отвечает за инфраструктуру спортивной организации. Таблица Branches хранит филиалы, таблица Room описывает помещения внутри филиала, а таблица Equipments содержит сведения об оборудовании. Между филиалами и сотрудниками реализованы отдельные связи для управляющих и персонала филиала. Для это используются промежуточные таблицы BranchAdmin и BranchStaffs, в которых составной первичный ключ формируется из идентификаторов филиала и сотрудника. Такая структура позволяет одному сотруднику работать с несколькими филиалами, а одному филиалу иметь несколько ответственных сотрудников.

Третий блок базы данных связан с услугами и расписанием. Таблица SportServices хранит справочник услуг, при этом поле Code является уникальным и позволяет однозначно идентифицировать услугу в системе. Стоимость услуги вынесена в таблицу ServicePrices, где сохраняется сумма, валюта и период действия цены. Это решение позволяет изменять стоимость услуги во времени без перезаписи старых данных. Для планирования занятий используется таблица TrainingSessions, где дополнительно указываются дата и статус проведения.

Четвертый блок отражает клиентские операции. Таблица Bookings связывает клиента с конкретной тренировочной сессией и хранит статус записи. Таблица Memberships используется для хранения абонементов клиентов: даты начала и окончания действия, общего числа посещений, остаток посещений и текущего статуса. Для упрощения оформления абонементов добавлена таблица MembershipTemplates. В ней заранее задаются параметры абонемента: филиал, услуга, цена, срок действия, лимит посещений и признаки активности. При покупке абонемента система может использовать выбранный шаблон, что уменьшает дублирования данных и делает процесс оформления более предсказуемым.

Для работы с изображениями в базе данных хранится только служебная информация о файле: идентификатор, имя объекта, имя исходного файла, тип содержимого, размер и дата создания. Сами изображения размещаются в объектном хранилище MinIO. Такой подход выбран потому, что хранение

бинарных файлов непосредственно в реляционной базе данных увеличивало бы ее объем и усложняло резервное копирование. При этом таблица Images позволяет связывать загруженные изображения с филиалами, помещениями, оборудованием, а также использовать их как аватары клиентов и сотрудников.

Целостность данных обеспечивается первичными и внешними ключами. Например, помещение не может существовать без филиала, оборудование связано с помещением, расписание связано с услугой, а запись клиента связана с конкретной тренировочной сессией. Для некоторых связей дополнительно настроено поведение при удалении. Так, удаление филиала не должно автоматически удалять управляющего, поэтому связи филиала с основным управляющим используется ограничение удаления. Для связи абонеента с шаблоном предусмотрено обнуление внешнего ключа, чтобы уже оформленный абонемент не терялся после удаления или изменения шаблона.

В проекте также используются перечислимые типы PostgreSQL. К ним относятся роли пользователей, статусы учетных записей, бронирований, абонементов, состояний тренировочных сессий, категории услуг, состояния помещений и оборудования. Использование перечислений ограничивает набор допустимых значений на уровне базы данных и уменьшает вероятность появления некоторых состояний, например несуществующего статуса записи или роли пользователя.

Изменения структуры базы данных выполняется через механизм миграции Entity Framework Core. Миграции позволяют пошагово фиксировать развитие системы: создание начальных таблиц, добавление клиентских сущностей, изображений, связей филиалов с сотрудниками, шаблонов абонементов и полей для работы с часовыми поясами. При запуске серверного приложения вызывается применение миграций, за счет чего рабочая база данных автоматически приводится к актуальному состоянию модели. Это упрощает развертывание системы в контейнерной среде и снижает риск между кодом приложения и структурной базы данных.

Отдельно в проекте реализовано хранилище аналитических событий на базе ClickHouse [22]. Оно не заменяет основную базу, а используется для задач аналитики, где требуется быстро выполнять агрегирующие запросы. В ClickHouse создается таблица `client_activity_events`, в которой сохраняются события активности клиентов: просмотр карточки филиала, открытие подробной информации, запись на тренировочную сессию и покупка абонемента. Для каждого события фиксируется время возникновения, дата, час, тип события, идентификатор клиента, филиала, услуги и дополнительные метаданные. Такое разделение позволяет не перегружать основную транзакционную базу аналитическими подсчетами.

В контейнерной конфигурации системы для PostgreSQL используется база `sport_management`, а данные сохраняются в отдельном постоянном томе. Это необходимо для сохранения информации после перезапуска контейнеров. Служебные данные фоновых задач HangFire также размещается в PostgreSQL, что позволяет планировщику восстанавливать состояние заданий после перезапуска серверной части.

Таким образом, разработанная база данных обеспечивает хранение всех ключевых данных системы. Реляционная структура PostgreSQL используется для основных бизнес-операций и поддержания целостности, MinIO применяется для хранения файлов изображений, а ClickHouse выделен для аналитических событий. Такое разделение хранилищ делает систему более устойчивой, расширяемой и удобной для дальнейшего сопровождения.

3.2 Реализация серверной части информационной системы

Серверная часть информационной системы управления спортивными организациями была реализована как отдельное веб-приложение `SportManagementSystem`. Она отвечает за обработку HTTP-запросов от клиентского приложения, выполнение бизнес-логики, проверку прав доступа, работу с базой данных, хранение изображений, выполнение фоновых задач и формирование аналитических данных. Такой подход позволяет отделить

пользовательский интерфейс от прикладной логики и обеспечить единый программный интерфейс для web и мобильного клиента.

Разработка серверной части выполнена на языке C# с использованием ASP.NET Core и платформы .NET 9 [23, 24]. В качестве основного способа взаимодействия с клиентской частью используется Rest API [25]. Все контроллеры имеют маршрут общего вида «v1/[controller]», что позволяет версионировать API и в дальнейшем развивать серверную часть без нарушения совместимости с уже существующим клиентом.

Архитектура серверной части представлена на рисунке 8.

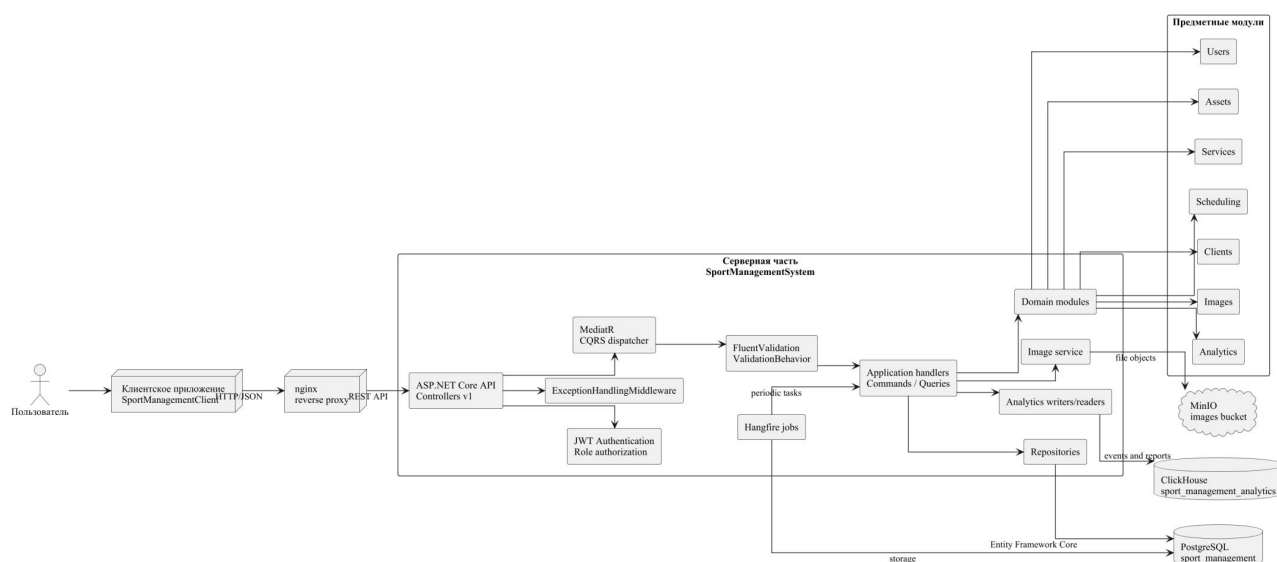


Рисунок 8 – Архитектура серверной части информационной системы

При реализации серверной части мной была выбрана модульная организация кода [26]. В проекте выделены предметные модули, каждый из которых отвечает за отдельную область функциональности. Такой подход упрощает сопровождение проекта: изменения, связанные с расписанием, клиентами или оборудованием, в основном локализуются внутри соответствующего модуля и не требуют переработки всей системы. Любое изменение бизнес-требований, например, введение нового типа абонементов или усложнение правил записи на занятия, реализуется с минимальным влиянием на смежные модули.

Основные модули серверной части приведены в таблице 8.

Таблица 8 – Модули серверной части информационной системы

Модуль	Назначения
Users	Регистрация, авторизация, учетные записи клиентов и сотрудников
Assets	Управление филиалами, помещениями и оборудованием
Services	Управление спортивными услугами и ценами
Schedules	Формирование расписания и тренировочных сессий
Clients	Записи клиентов, абонементы и шаблоны абонементов
Images	Загрузка, получение и удаление изображений
Analytics	Сбор событий активности и получение аналитических показателей

Внутри модулей используется разделение на доменный, прикладной, инфраструктурный слои и контракты обмена данными. Доменный слой содержит сущности, перечисления и интерфейсы репозитория. Прикладной слой содержит команды, запросы, обработчики и валидаторы. Инфраструктурный слой реализуется для описания входных запросов и выходных ответов API. Такое разделение позволило уменьшить связность между контроллерами, бизнес-логикой и механизмом хранения данных.

Для обработки пользовательских сценариев используется библиотека MediatR и подход CQRS. Команды применяются для операций, изменяющих состояние системы, например регистрации пользователя, создания филиала, добавления услуги, оформление абонемента или записи на занятие. Запросы используются для получения данных: списка филиалов, информации о клиенте, расписания, тренировочных сессий и аналитики. Контроллеры при таком подходе остаются компактными: они принимают HTTP-запрос, формируют команду или запрос и передают его в соответствующий обработчик через MediatR. Общий порядок обработки запроса показан на рисунке 9.

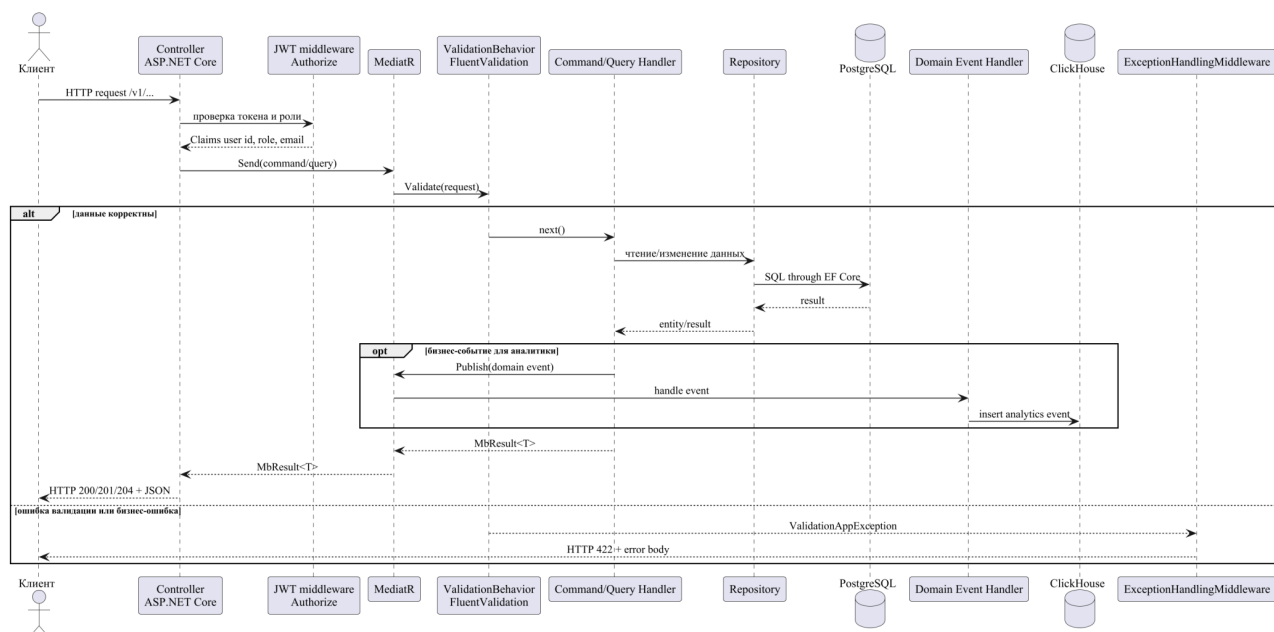


Рисунок 9 – Последовательность обработки HTTP-запроса на сервере

Перед выполнением прикладной логики входные данные проходят проверку через FluentValidation [27]. Для этого в конвейер MediatR добавлен ValidationBehavior, который находит валидаторы для конкретного сообщения, выполняют проверку и при наличии ошибок формирует исключение валидации. Благодаря этому правила проверки не смешиваются с кодом контроллеров, а ошибки возвращаются клиенту в едином формате.

В серверной части реализована аутентификация и авторизация на основе JWT [28]. При входе пользователь передает электронную почту и пароль, после чего сервер учетную запись, сравнивает пароль с сохраненным хэшем и формирует токены доступа. Для хранения паролей используется шифрование BCrypt, поэтому исходные пароли пользователей не сохраняются в базе данных. В JWT помещаются основные сведения, необходимые для дальнейшей проверки доступа: email, роль, идентификатор пользователя и идентификатор учетной записи.

Для ограничения доступа к защищенным операциям используются атрибуты «Authorize» и проверка ролей. Например, создание филиалов и управление рядом административных сущностей доступно только управляющему, операции клиента ограничены ролью «Client». При этом

публичные данные, такие как список филиалов, информация об услугах или изображениях, могут открываться без авторизации через атрибут «AllowAnonymous».

Основные группы API представлены в таблице 9.

Таблица 9 – Основные группы маршрутов серверной части

Группа API	Назначение
v1/authentication	Регистрация клиентов, управляющих и сотрудников, вход в системы
v1/clients	Получение данных клиентов, личного профиля и загрузка аватара
v1/staffs	Получение данных сотрудников, личного профиля и загрузка аватара
v1/branches	Создание и получение филиалов, работа с изображениями филиалов
v1/rooms	Создание, получение и удаление помещений, работа с изображениями помещений
v1/equipments	Создание, получение и удаление оборудования, работа с изображениями оборудования
v1/sportservices	Создание, получение и удаление спортивных услуг
v1/serviceprices	Создание и получение цен на услуги
v1/serviceschedules	Создание, получение и удаление расписания
v1/trainingssessions	Создание, получение и удаление тренировочных сессий
v1/bookings	Запись клиента на тренировочную сессию и просмотр занятий
v1/memberships	Оформление и просмотр абонеента
v1/membershiptemplates	Создание, получение и удаление шаблонов абонеента

Продолжение таблицы 9

Группа API	Назначение
v1/images	Загрузка, получение и удаление файлов изображений
v1/analytics	Получение аналитики и фиксация событий активности

Для возврата результатов операций используется единый объект «MbResult<T>». Если операция завершилась успешно, контроллер возвращает данные и соответствующий HTTP-статус. Если возникла ошибка, клиент получает объект с названием ошибки, кодом статуса, описанием и деталями. Дополнительно в проекте реализован «ExceptionHandlerMiddleware», который перехватывает необработанные исключения и приводит их к единому JSON-формату. Это упрощает обработку ошибок на клиентской стороне и делает поведение API более предсказуемым.

Отдельное внимание уделено работе с изображениями. При загрузке изображения сервер проверяет размер файла, при необходимости сжимает JPEG и PNG изображения с помощью ImageSharp, сохраняет объект в MinIO и фиксирует запись в базе данных. Если во время сохранения возникает ошибка, выполняется откат операции, чтобы не оставлять несогласованные данные.

Для аналитических функций в проект добавлен модуль Analytics. Он фиксирует события активности клиента: просмотр карточки филиала, открытие подробной информации, запись на тренировочную сессию и покупку абонемента. Такие события записываются в ClickHouse, где они используются для построения отчетов по филиалам и активности пользователей. Основные бизнес данные при этом продолжают храниться в PostgreSQL, а аналитическая нагрузка выносится в отдельное хранилище.

В серверной части также используется Hangfire для выполнения фоновых задач [29]. С его помощью настроена периодическая задача TrainingSessionStatusDispatcher, которая запускается каждые пять минут и

обновляет статусы тренировочных сессий. Это позволяет не привязывать служебную обработку расписания к действиям пользователя и выполнять ее автоматически.

Конфигурация серверной части вынесена в файлы настроек приложения. В них задаются строки подключения к PostgreSQL и ClickHouse, параметры JWT, настройки MinIO и часовой пояс по умолчанию. При запуске приложения регистрируются все необходимые сервисы: контекст базы данных, репозитории, MediatR, FluentValidation, JWT-аутентификация, MinIO, ClickHouse, Hangfire, Swagger [30] и middleware обработки ошибок.

Для развертывания серверной части используется Docker Compose. В состав серверного контура входят контейнер API-приложения, PostgreSQL, MinIO, ClickHouse и nginx. Nginx выполняет роль обратного прокси, PostgreSQL хранит основные данные системы и служебные данные Hangfire, MinIO используется для хранения файлов изображений, а ClickHouse – для аналитических событий. За счет контейнеризации окружение запускается воспроизводимо и не зависит от ручной настройки отдельных сервисов на рабочем компьютере.

Таким образом, серверная часть информационной системы реализует все основные бизнес-сценарии проекта: регистрацию и вход пользователей, управление филиалами и ресурсами, настройку услуг и расписания, оформление абонементов, запись клиентов на занятия, хранение изображений и сбор аналитики. Выбранная структура серверной части обеспечивает разделение ответственности между модулями, упрощает сопровождение кода и создает основу для дальнейшего расширения функциональности системы. Исходный код серверной части размещен в открытом репозитории [31].

3.3 Реализация клиентской части информационной системы

Клиентская часть информационной системы управления спортивными организациями реализована в проекте SportManagementClient. Она предназначена для взаимодействия пользователя с основными функциями системы: просмотра филиалов, получения информации о помещениях, услугах и

расписании, регистрации и авторизации, покупки абонементов, записи на занятия, управления филиалами и просмотра аналитики. Клиентское приложение обращается к серверной части через REST API и отображает данные в удобном для пользователя виде.

Для разработки клиентской части был выбран React Native с использованием TypeScript [32]. Такой выбор позволяет использовать единую кодовую базу для нескольких платформ: Web, Android и IOS. Для web-версии применяется «react-native-web» и сборка через Webpack, а для мобильных платформ используется стандартная мобильная инфраструктура React Native. Это уменьшает объем дублирования кода и позволяет развивать интерфейс системы как единое приложение.

Структура клиентской части приложения построена по feature-ориентированному принципу. В каталоге «src/app» размещены корневой компонент приложения и основные экранные контейнеры. В каталоге «src/features» находятся функциональные области: авторизация, работа с филиалами, клиентские операции и профиль пользователя. Общие компоненты интерфейса, HTTP-клиент, настройки API и вспомогательные функции вынесены в каталог «src/shared». Данные текущей сессии пользователя находятся в каталоге «src/entities/session». Основные части клиентского приложения приведены в таблице 10.

Таблица 10 – Основные части клиентского приложения

Часть приложения	Назначение
app	Корневой компонент, маршрутизация, общая навигационная панель и подключение экранов
features/auth	Вход, регистрация клиента и регистрация управляющих
features/branches	Просмотр филиалов, карточка филиала, управление филиалами, помещениями и оборудованием

Продолжение таблицы 10

Часть приложения	Назначение
features/client	Покупка абонементов, запись на занятия, просмотр клиентской активности
features/profile	Профиль пользователя, загрузка аватара, управление шаблонами абонементов и аналитика
shared/api	Единый HTTP-клиент для общения с серверной частью
shared/ui	Переиспользуемые элементы интерфейса: кнопки, поля ввода, контейнер экрана, уведомления
entities/session	Сохранение, восстановление и проверка пользовательской сессии

Корневым элементом приложения является компонент «App.tsx». В нем реализована загрузка сохраненной сессии, проверка срока действия токена, переключение маршрутов, отображение верхней панели навигации и подключение нужного экрана. Для web-версии используются History API браузера, поэтому маршруты вида «/login», «profile», «branches/{id}» и «/admin-branches/{id}» отражаются в адресной строке. Для мобильных платформ маршрут хранится во внутреннем состоянии приложения, что позволяет использовать тот же набор экранов без отдельной реализации навигации.

Главный экран приложения отображает список филиалов спортивных организаций. Для каждого филиала показывается название, адрес, количество помещений и фотографии, если они были загружены управляющим. Также реализовано отображение филиалов на карте: в web-версии используется интерактивная карта Яндекса [33], а на мобильных платформах формируется статическое изображение карты. При просмотре карточек филиалов и открытии подробной информации клиентская часть отправляет события в модуль аналитики, что позволяет управляющему видеть активных пользователей.

Карточка филиала содержит подробную информацию о выбранном филиале: фотографии, адрес, помещения, оборудование, список услуг, расписание, сотрудников, шаблоны абонементов, и тренировочные сессии. Для расписания предусмотрено несколько вариантов отображения, включая список и календарный режим. Если пользователь авторизован как клиент, на этом экране он может выбрать шаблон абонемента и оформить покупку, а также выбрать тренировочную сессию и записаться на занятие.

Функции авторизации реализованы в экранах «LoginScreen» и «RegisterScreen». При входе пользователь указывает электронную почту и пароль, после чего клиентская часть отправляет запрос на сервер и получает JWT-токен с датой истечения срока действия. После успешного входа токен сохраняется в объекте сессии вместе с областью роли пользователя. Для регистрации реализованы формы клиента и управляющего. В формах используются локальные проверки обязательных полей, пароля, даты рождения и телефона, что снижает количество некорректных запросов к серверу.

Сессия пользователя сохраняется по-разному в зависимости от платформы. На Android и IOS используется «AsyncStorage», а в web-версии – «localStorage». При запуске приложения сессия восстанавливается, проверяется ее структура и срок действия. Если токен просрочен или сервер возвращает статус 401, приложение удаляет сессию, переводит пользователя на экран входа и показывает уведомление. Такое поведение предотвращает работу с устаревшими токенами и делает авторизацию более надежной.

В приложении используется разделение интерфейса по ролям. Неавторизованный пользователь может просматривать публичную информацию о филиалах, услугах и расписании. Клиент получает доступ к профилю, покупке абонементов, записи на занятия и экрану собственной активности. Управляющий получает доступ к управлению филиалами, сотрудниками, помещениями, оборудованием, услугами, расписанием, тренировочными сессиями, шаблонами абонементов и аналитикой. Набор пунктов верхней навигации формируется на основе роли пользователя, полученной из JWT.

Личный кабинет клиента содержит сценарии покупки абонемента и записи на тренировочную сессию. Пользователь выбирает филиал или услугу или готовый шаблон абонемента, после чего клиентская часть отправляет запрос на создание абонемента. Для записи используется выбранная тренировочная сессия. Отдельный экран «ClientActivityScreen» показывает историю абонементов и записей, а также сводные показатели: количество абонементов, все записи и предстоящие занятия.

Профиль пользователя реализован общим экраном «ProfileScreen». Он загружает сведения о текущем пользователе и позволяет обновить аватар. Перед отправкой аватара web-версия может сжать изображение на стороне клиента, если файл превышает допустимый размер. Для этого используется canvas и преобразование изображения в JPEG. Такой подход уменьшает объем передаваемых данных и снижает вероятность ошибки загрузки.

Экран аналитики предназначен для управляющих. Он получает агрегированные данные по филиалам из серверной части и отображает показатели просмотров карточек, открытий подробной информации, записей на занятия и покупок абонементов. Для визуализации трендов используется «react-native-svg», что позволяет строить графики без привязки к web-only библиотекам. Пользователь может переключать метрики и выбрать детализацию тренда по дням, часам или десятиминутным интервалам.

Для обмена данными с сервером единый HTTP-клиент реализован в «shared/api/httpClient.ts». Он формирует URL запроса, добавляет заголовок «Authorization» при наличии токена, сериализует JSON-тело запроса, обрабатывает пустые ответы, текст, JSON и файлы. Если сервер возвращает ошибку, HTTP-клиент приводит ее к единому объекту с кодом статуса, сообщением, деталями и возможным списком ошибок валидации. Это позволяет экранам не дублировать низкоуровневую обработку ответов сервера.

В интерфейсе используются общие компоненты «AppButton», «AppInput», «ScreenContainer», «AppToast» и «GoogleIcon». Это позволяет сохранить единый визуальный стиль на разных экранах и уменьшить повторение кода. Для

большинства экранов предусмотрены состояния загрузки, ошибки и успешное выполнение операции. Всплывающие уведомления используются для сообщений о входе, выходе, истечении сессии, успешной загрузке изображения и других действиях пользователя.

Клиентская часть учитывает адаптивность интерфейса. В экранах применяются «useWindowDimensions», за счет чего интерфейс меняет расположение элементов на широких и узких экранах. Например, на desktop-ширине некоторые блоки располагаются в несколько колонок, а на мобильных устройствах выстраиваются вертикально. Это важно, так как приложение должно быть пригодным как для web-версии, так и для мобильного использования.

Таким образом, клиентская часть информационной системы реализует полноценный пользовательский интерфейс для разных категорий пользователей. Она обеспечивает публичный просмотр филиалов, авторизацию, личный кабинет, управление объектами спортивными организациями, загрузку изображений и просмотр аналитики. Использование React Native и общей feature-структуры позволило создать единое приложение для нескольких платформ и связать его с серверной частью через устойчивый слой API-взаимодействия. Исходный код клиентской части размещен в открытом репозитории [34].

3.4 Реализация интерфейса приложения

Интерфейс информационной системы управления спортивными организациями был призван с учетом разных категорий пользователей: посетителя, клиента, администратора филиала и управляющего. При разработке экранов я придерживался единого подхода: основные действия выделены синим цветом, данные представлены в карточках, а формы ввода сгруппированы по смыслу. Это позволяет пользователю быстрее понимать назначение страницы и выполнять нужные действия без лишних переходов.

Так как приложение содержит как клиентское, так и административные сценарии, интерфейс был разделен по ролям. Неавторизованный пользователь

может просматривать филиалы и переходить к регистрации. Клиент получает доступ к покупке абонементов и записи на занятия. Управляющий работает с филиалами, помещениями, оборудованием, услугами, расписанием, сотрудниками и шаблонами абонементов.

Ниже приведены основные экраны приложения. После каждого рисунка указано назначение элементов, отмеченных на скриншотах стрелками с номерами.

Экран авторизации пользователя показан на рисунке 10.

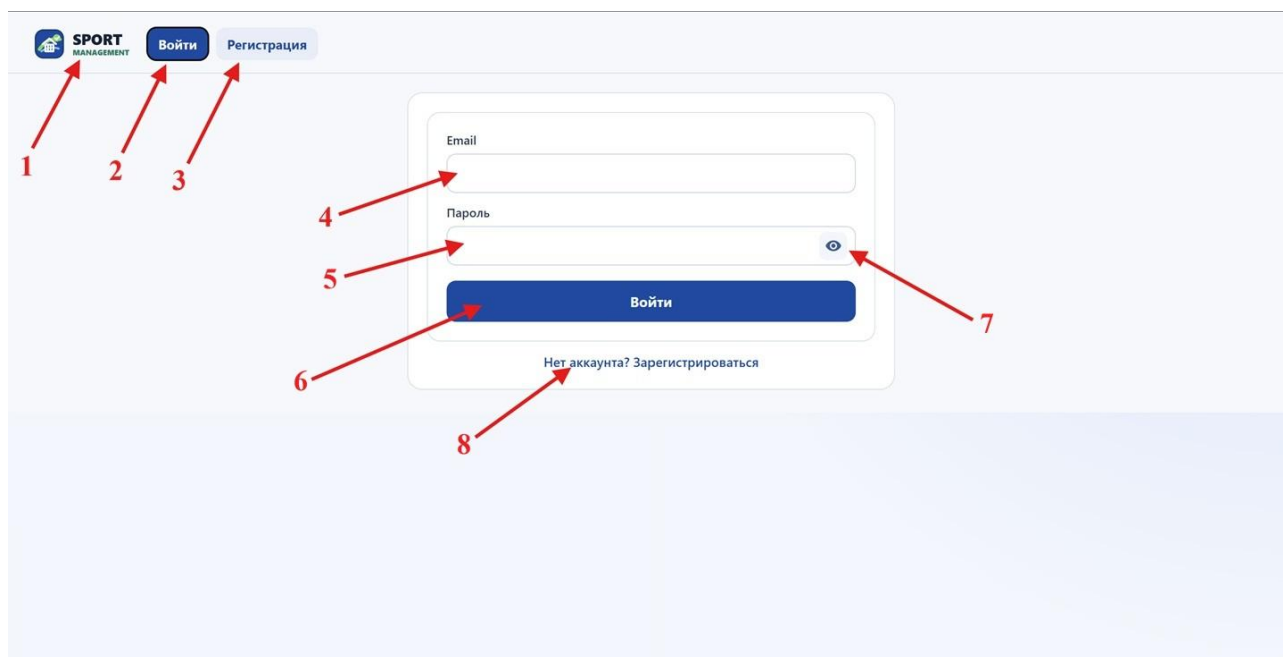


Рисунок 10 – Экран авторизации пользователя

На рисунке 10 обозначены следующие элементы интерфейса:

- 1 – логотип приложения;
- 2 – вкладка входа;
- 3 – вкладка регистрации;
- 4 – поле для ввода электронной почты;
- 5 – поле для ввода пароля;
- 6 – кнопка отправки формы входа;
- 7 – кнопка показа или скрытия пароля;
- 8 – ссылка для перехода к регистрации.

Экран регистрации пользователя представлен на рисунке 11, обозначены:

- 1 – вкладка регистрации клиента;
- 2 – вкладка регистрации администратора филиала;
- 3 – поле для ввода имени;
- 4 – поле для ввода отчества;
- 5 – поле для ввода телефона;
- 6 – поле для ввода пароля;
- 7 – поле для ввода фамилии;
- 8 – поле для выбора даты рождения;
- 9 – поле для ввода электронной почты;
- 10 – поле для повторного ввода пароля;
- 11 – кнопка создания аккаунта.

The screenshot shows a registration form titled 'SPORT MANAGEMENT' with tabs for 'Войти' (Login) and 'Регистрация' (Registration). The registration form has two main sections: 'Клиент' (Client) and 'Администратор филиала' (Branch administrator). The 'Клиент' section includes fields for 'Имя' (Name), 'Отчество (необязательно)' (Middle name, optional), 'Телефон' (Phone), 'Пароль' (Password), and 'Повторите пароль' (Repeat password). The 'Администратор филиала' section includes fields for 'Фамилия' (Surname), 'Дата рождения' (Date of birth), and 'Email'. A 'Создать аккаунт' (Create account) button is at the bottom. Red arrows with numbers 1 through 11 point to specific elements: 1 points to the 'Клиент' tab, 2 to the 'Администратор филиала' tab, 3 to the 'Имя' field, 4 to the 'Отчество' field, 5 to the 'Телефон' field, 6 to the 'Пароль' field, 7 to the 'Фамилия' field, 8 to the 'Дата рождения' field, 9 to the 'Email' field, 10 to the 'Повторите пароль' field, and 11 to the 'Создать аккаунт' button.

Рисунок 11 – Экран регистрации пользователя

Главная страница приложения с картой и списком филиалов показана на рисунке 12, на нем обозначены:

- 1 – кнопка увеличения масштаба карты;
- 2 – кнопка уменьшения масштаба карты;
- 3 – кнопка открытия карты в Яндекс картах;

4 – кнопка открытия подробной информации;

5 – ссылка на филиал в Яндекс картах.

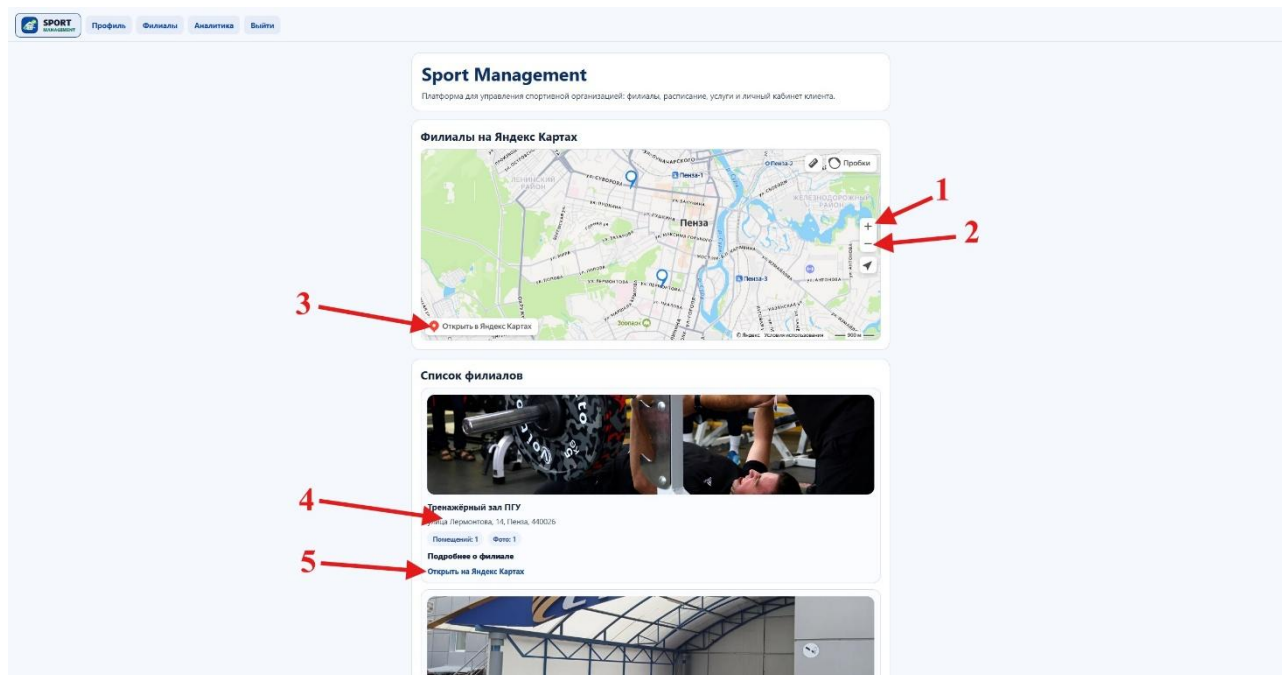


Рисунок 12 – Главная страница приложения

Первая часть подробной информации о филиале представлена на рисунке 13 и имеет следующие элементы:

1 – кнопка возврата к списку филиалов;

2 – пункт перехода к странице «Мои услуги»;

3 – кнопка открытия филиала в Яндекс Картах;

4 – кнопка увеличения фотографии.

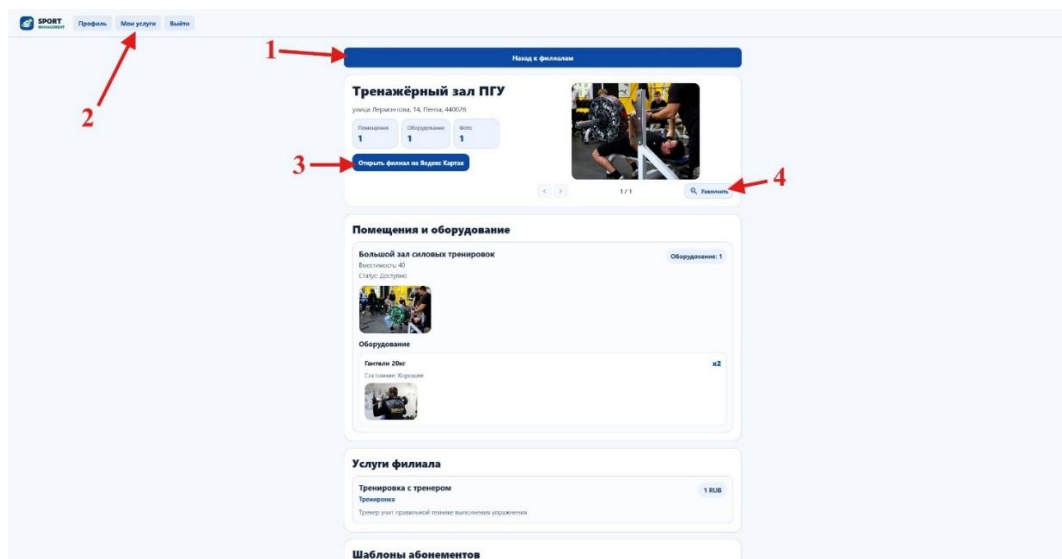


Рисунок 13 – Подробная информация о филиале, часть 1

Блок выбора и оформления абонемента на странице филиала показан на рисунке 14, а обозначены там следующие элементы:

- 1 – карточка абонемента;
- 2 – кнопка оформления абонемента.

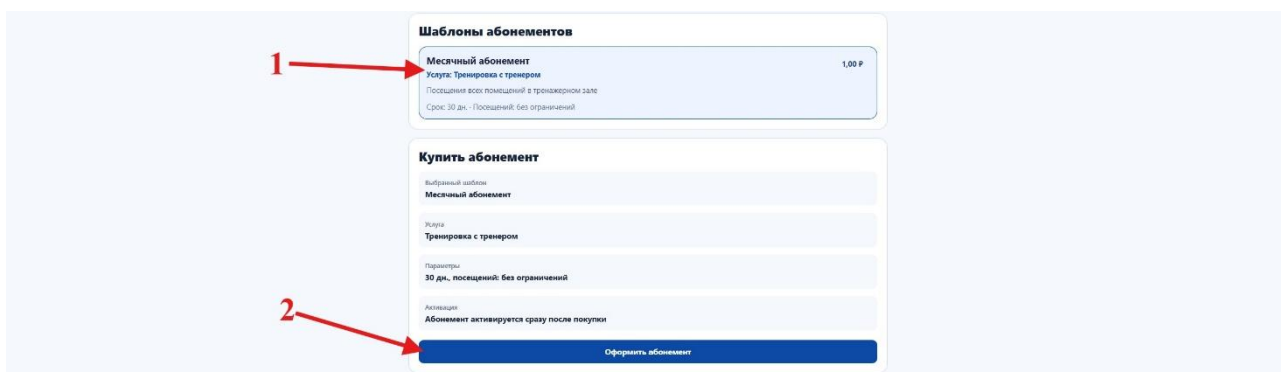


Рисунок 14 – Подробная информация о филиале, часть 2

Часть страницы филиала с расписанием занятий представлена на рисунке 15, с обозначенными элементами:

- 1 – список доступных занятий;
- 2 – карточка занятия с датой, временем, тренером и статусом;
- 3 – кнопка записи на занятие;
- 4 – переключатель режима отображения расписания.

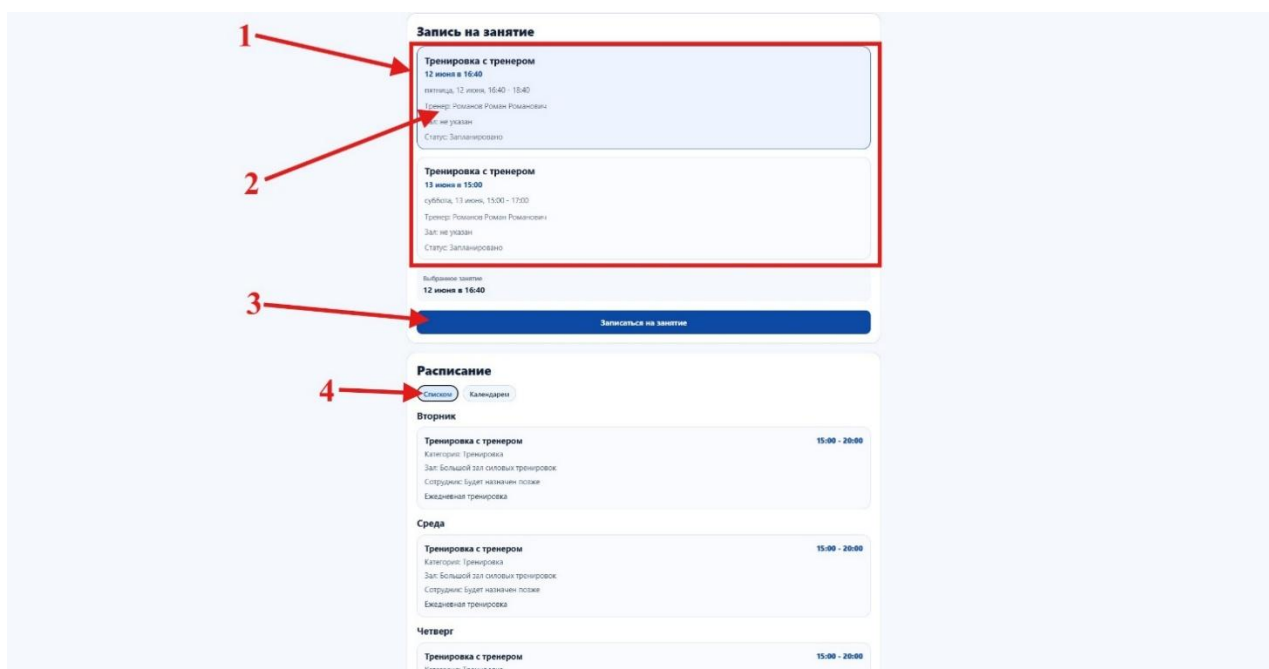


Рисунок 15 – Подробная информация о филиале, часть 3

Страница клиента «Мои услуги» показана на рисунке 16, со следующими элементами:

- 1 – кнопка обновления данных;
- 2 – список абонементов клиента;
- 3 – список записей на занятия.

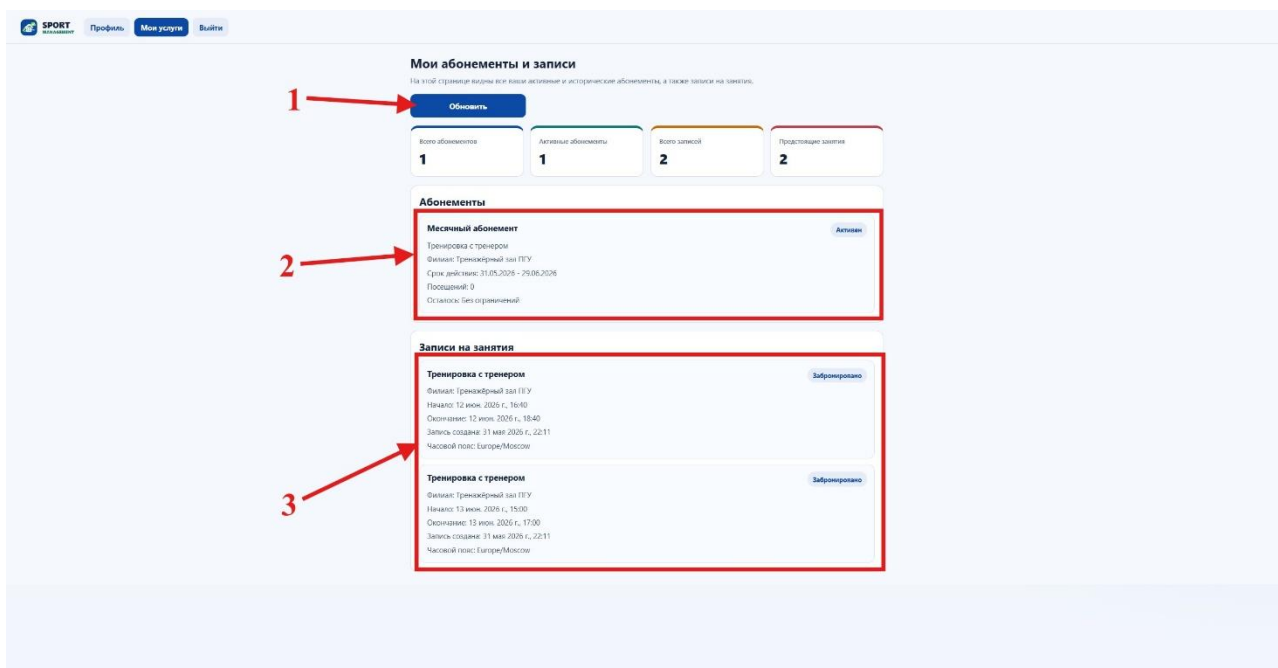


Рисунок 16 – Страница абонементов и записей клиента

Интерфейс профиля текущего пользователя приведен на рисунке 17, имеет несколько элементов:

- 1 – кнопка добавления изображения профиля;
- 2 – кнопка выхода из системы.

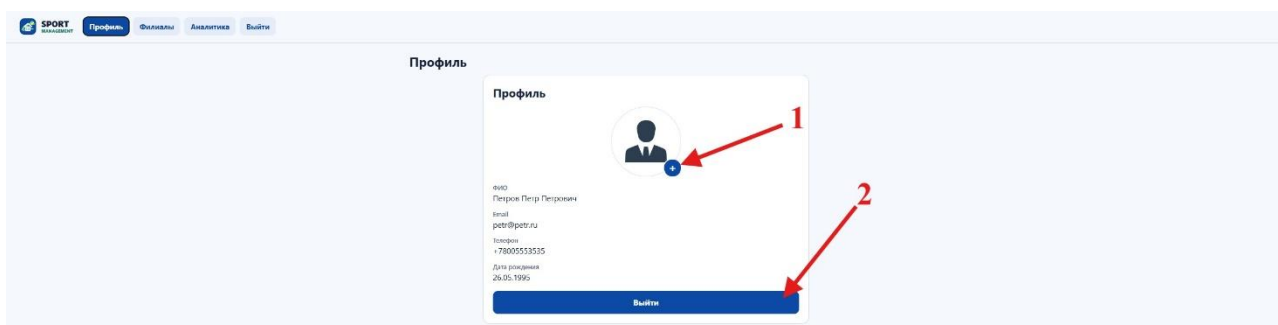


Рисунок 17 – Интерфейс управления профилем пользователя

Экран выбора филиала управляющим показан на рисунке 18, где под цифрой 1 обозначен список филиалов администратора. У каждого филиала в списке показывается минимальный набор необходимый для идентификации филиала: название, адрес и координаты.

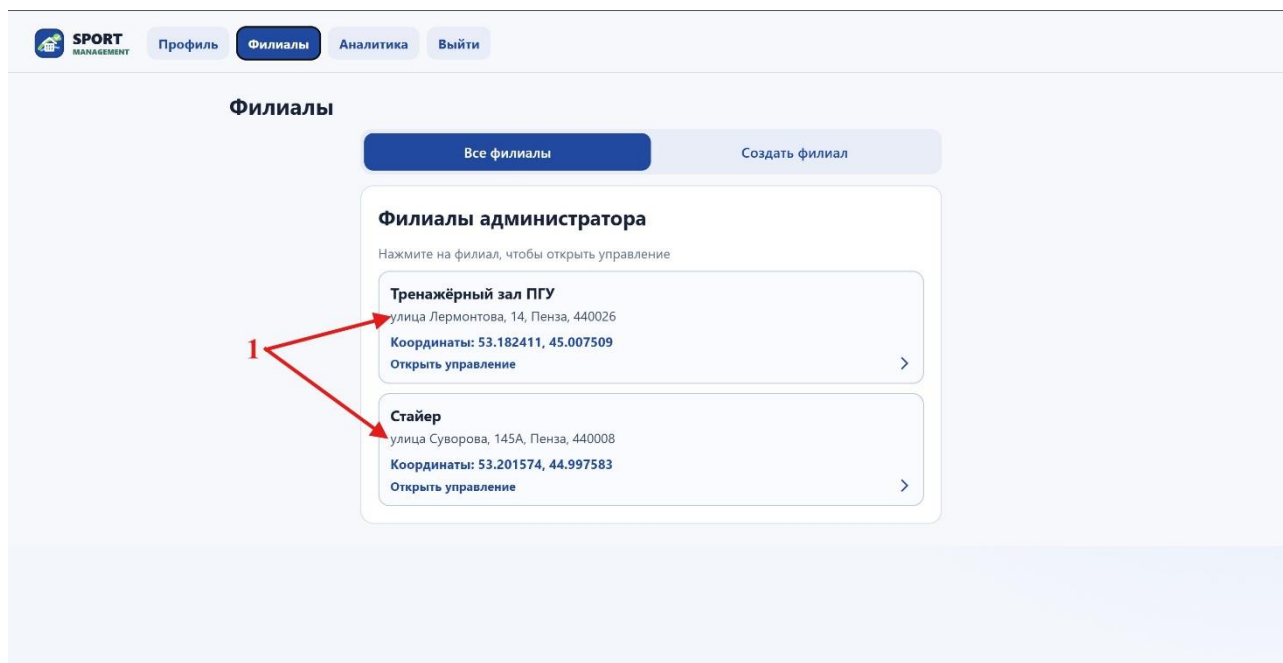


Рисунок 18 – Экран выбора филиала управляющим

Форма создания нового филиала представлена на рисунке 19, в котором обозначены:

- 1 – логотип приложения;
- 2 – пункт перехода к профилю;
- 3 – пункт перехода к филиалам;
- 4 – пункт перехода к аналитике;
- 5 – пункт выхода из системы;
- 6 – вкладка создания филиала;
- 7 – поле для ввода названия филиала;
- 8 – поле для ввода адреса филиала;
- 9 – поле для ввода широты;
- 10 – поле для ввода долготы;
- 11 – кнопка создания филиала.

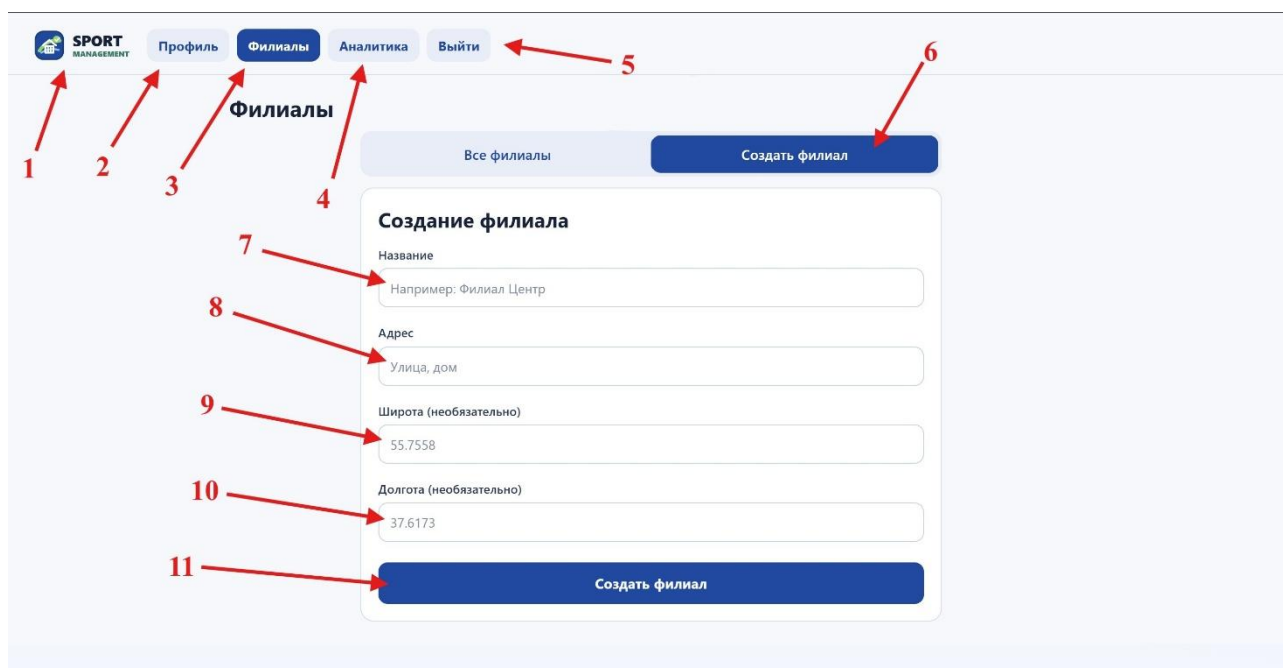


Рисунок 19 – Экран создания филиала

Основной экран управления филиалом с вкладками разделов показан на рисунке 20, который имеет следующие элементы:

- 1 – ссылка возврата к списку филиалов;
- 2 – кнопка добавления фотографий филиала;
- 3 – кнопка удаления фотографии филиала;
- 4 – вкладка «Помещения»;
- 5 – вкладка «Услуги»;
- 6 – вкладка «Расписание»;
- 7 – вкладка «Занятия»;
- 8 – вкладка «Сотрудники»;
- 9 – вкладка «Абонементы».

Экран управления филиалом объединяет основные административные разделы выбранного объекта. Благодаря вкладкам «Помещения», «Услуги», «Расписание», «Занятия», «Сотрудники» и «Абонементы» администратор может быстро переключаться между задачами: от управления пространством и персоналом до контроля за расписанием и продажами абонементов.

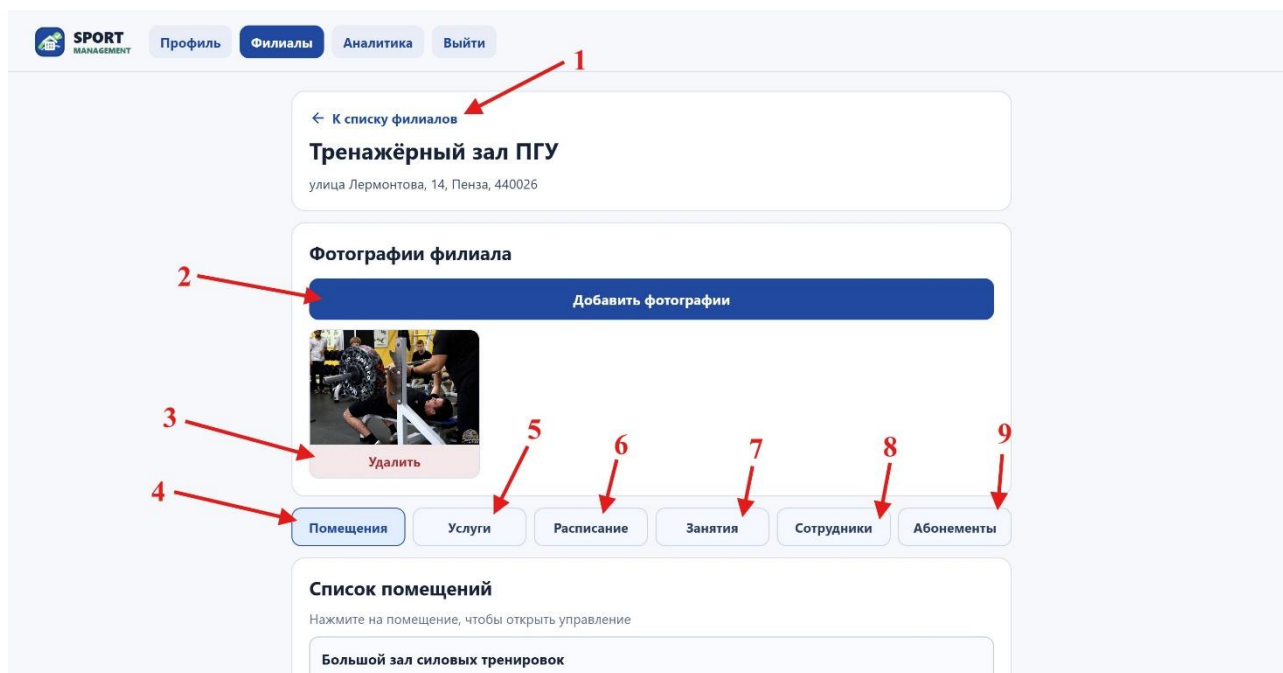


Рисунок 20 – Экран управления филиалом, часть 1

Фрагмент управления помещениями филиала приведен на рисунке 21, со следующими элементами:

- 1 – кнопка открытия помещения;
- 2 – кнопка удаления помещения;
- 3 – поле для ввода названия помещения;
- 4 – поле для ввода вместимости;
- 5 – кнопка добавления помещения.

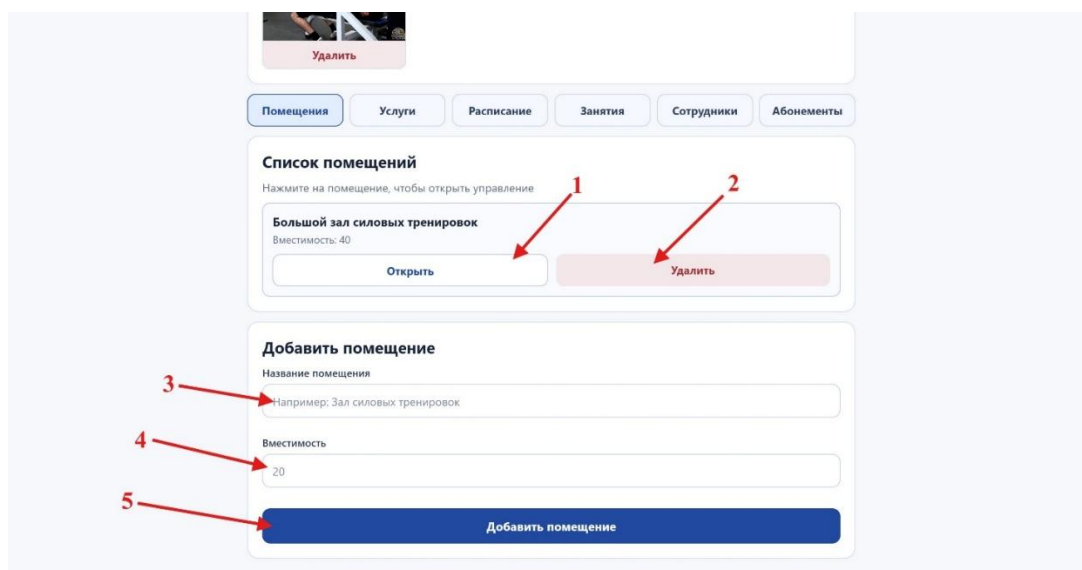


Рисунок 21 – Экран управления филиалом, часть 2

Интерфейс управления услугами филиала показан на рисунке 22, который имеет следующие элементы:

- 1 – список услуг филиала;
- 2 – кнопка удаления услуги;
- 3 – поле для ввода названия услуги;
- 4 – поле для ввода описания услуги;
- 5 – выбор категории услуги;
- 6 – поле для ввода цены;
- 7 – выбор валюты;
- 8 – кнопка сохранения услуги;
- 9 – кнопка отмены действия.

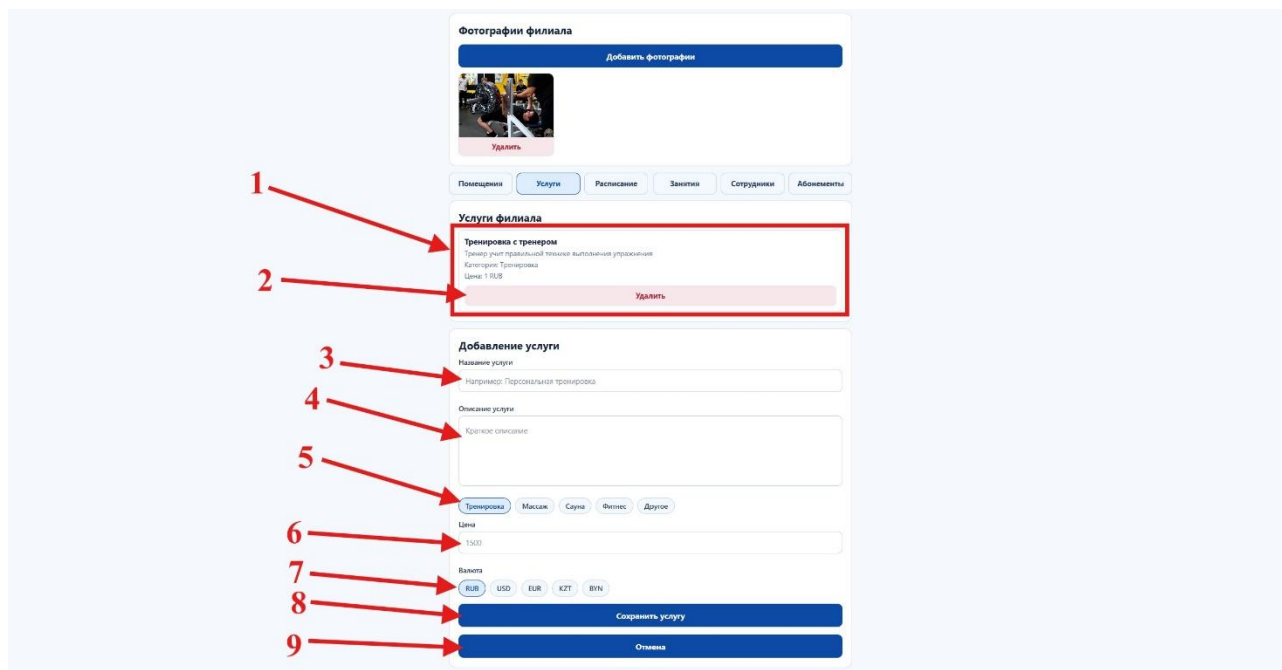


Рисунок 22 – Интерфейс управления услугами филиала

Управление расписанием в режиме списка и добавления представлено на рисунке 23, где обозначены:

- 1 – кнопка удаления слота расписания;
- 2 – список слотов расписания;
- 3 – выбор услуги;
- 4 – выбор помещения;

- 5 – выбор дней недели;
- 6 – поле для ввода времени начала;
- 7 – поле для ввода времени окончания;
- 8 – поле для ввода комментария;
- 9 – кнопка добавления в расписание.

Такая структура минимизирует ошибки при ручном вводе и упрощает формирование регулярного расписания на несколько дней вперед. Данный интерфейс позволяет администратору филиала гибко управлять временными слотами в интуитивно понятном интерфейсе.

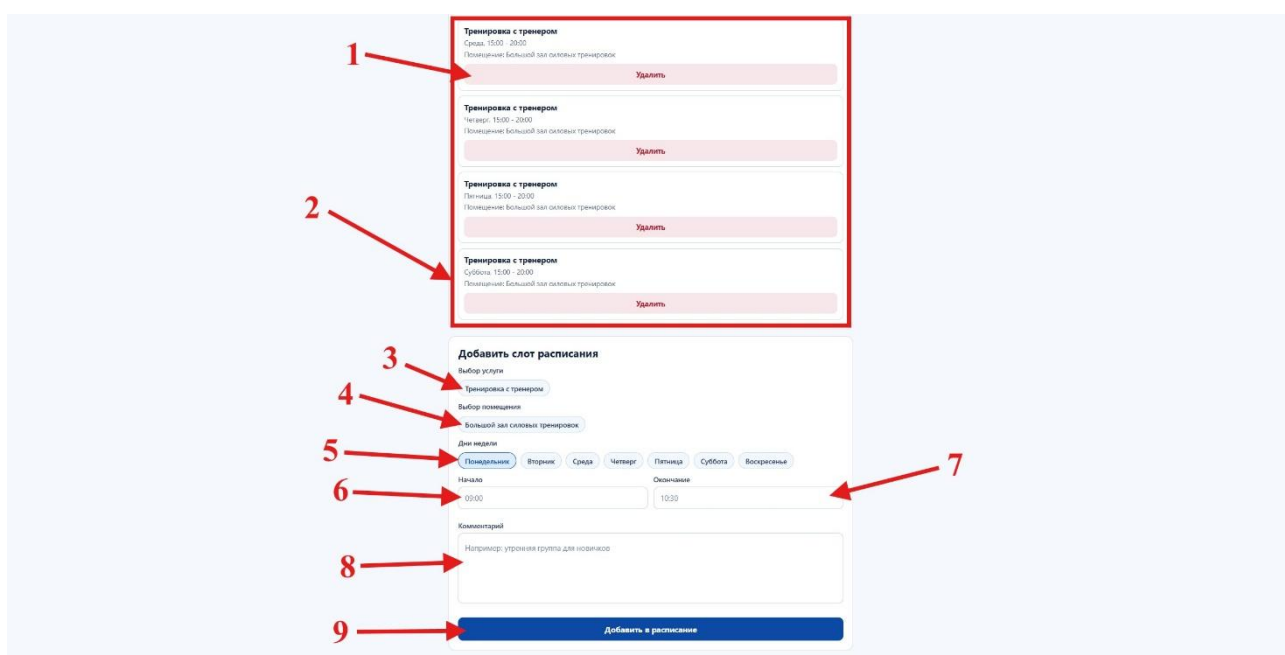


Рисунок 23 – Управление расписанием в режиме списка и добавления

Календарный режим просмотра расписания показан на рисунке 24, с обозначенными:

- 1 – переключатель просмотра всей недели;
- 2 – переключатель просмотра одного дня;
- 3 – календарная сетка расписания.

Календарный режим предоставляет функционал для визуального анализа распределения учебных мероприятий в соответствии с днями недели и временными интервалами.

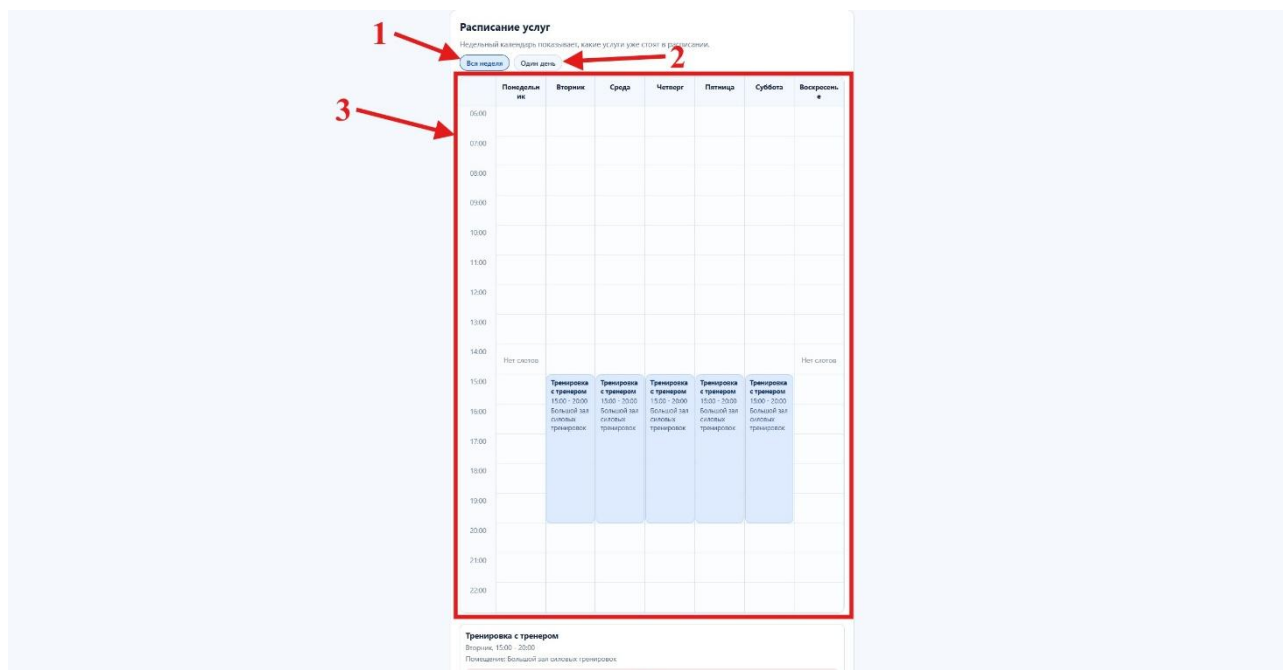


Рисунок 24 – Управление расписанием в календарном режиме

Экран управления сотрудниками филиала приведен на рисунке 25, со следующими обозначениями:

- 1 – список сотрудников филиала;
- 2 – кнопка удаления сотрудника;
- 3 – поле для ввода имени;
- 4 – поле для ввода фамилии;
- 5 – поле для ввода отчества;
- 6 – поле для выбора даты рождения;
- 7 – поле для ввода телефона;
- 8 – поле для ввода электронной почты;
- 9 – поле для ввода пароля;
- 10 – выбор роли сотрудника;
- 11 – кнопка регистрации сотрудника.

Экран сотрудников используется для добавления персонала филиала и назначения ролей в системе. Такой подход обеспечивает централизованное управление кадрами, разграничение доступа и быструю адаптацию нового персонала к работе в системе.

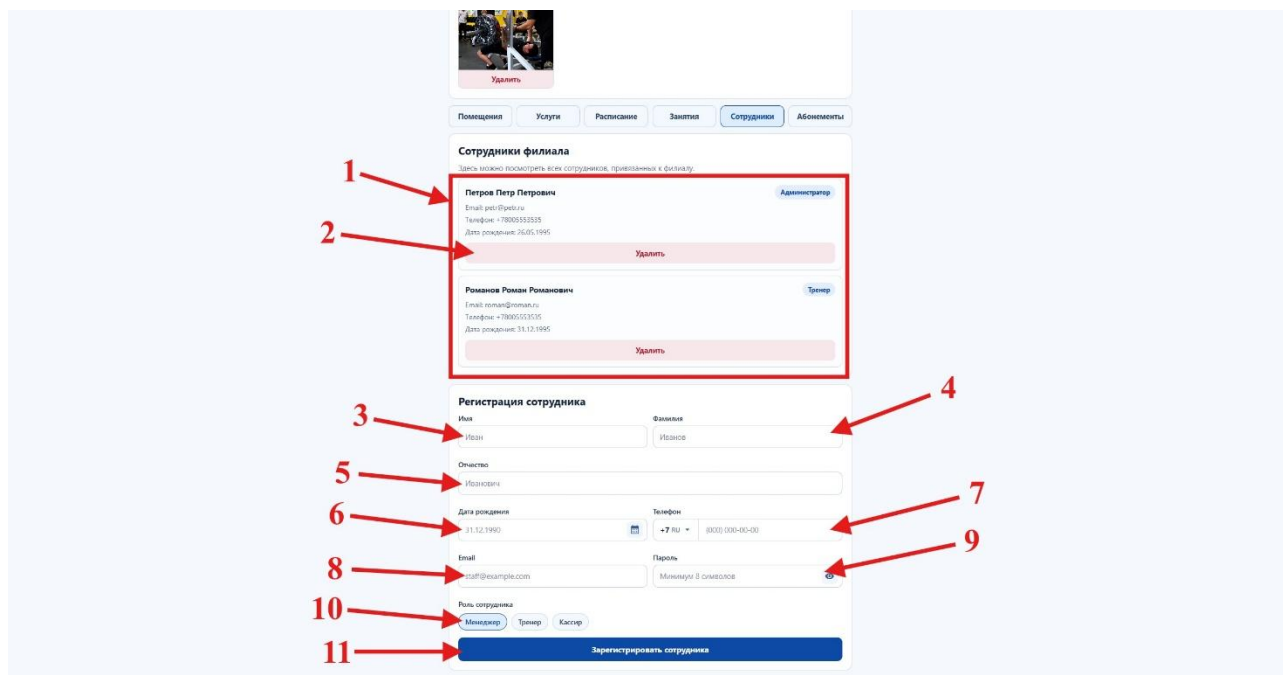


Рисунок 25 – Интерфейс управления сотрудниками филиала

Интерфейс управления шаблонами абонементов показан на рисунке 26, который имеет следующие элементы:

- 1 – список шаблонов абонементов;
- 2 – кнопка удаления абонемента;
- 3 – выбор услуги филиала;
- 4 – поле для ввода названия абонемента;
- 5 – поле для ввода описания абонемента;
- 6 – поле для ввода цены;
- 7 – выбор валюты;
- 8 – поле для ввода срока действия;
- 9 – поле для ввода лимита посещений;
- 10 – кнопка создания абонемента.

Шаблоны абонементов задают условия покупки услуги, которые затем отображаются клиенту на странице филиала. Благодаря такой структуре администратор может быстро создавать разные типы абонементов под одну услугу и управлять ими без дублирования информации.

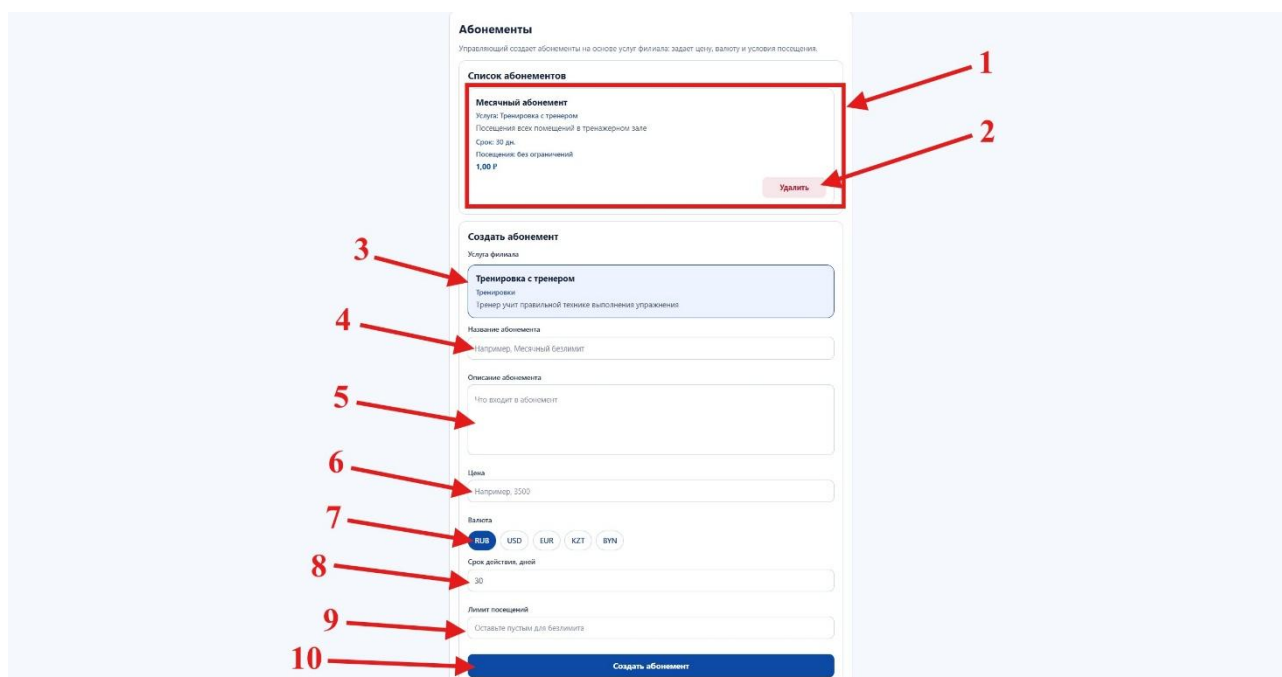


Рисунок 26 – Интерфейс управления шаблонами абонементов

Интерфейс создания и просмотра тренировочных занятий представлен на рисунке 27, где обозначены:

- 1 – список тренировочных занятий;
- 2 – кнопка удаления занятия;
- 3 – выбор услуги;
- 4 – выбор тренера;
- 5 – выбор слота расписания;
- 6 – поле выбора даты и времени начала;
- 7 – поле выбора даты и времени окончания;
- 8 – кнопка создания занятия.

Занятие представляет собой конкретное мероприятие, на которое клиент может зарегистрироваться через веб-сайт филиала. Данный метод обеспечивает возможность гибкого формирования как регулярных занятий (в соответствии с расписанием слотов), так и разовых мероприятий (например, мастер-классов или открытых тренировок). Это упрощает процесс регистрации клиентов и позволяет эффективно контролировать загрузку тренерского состава.

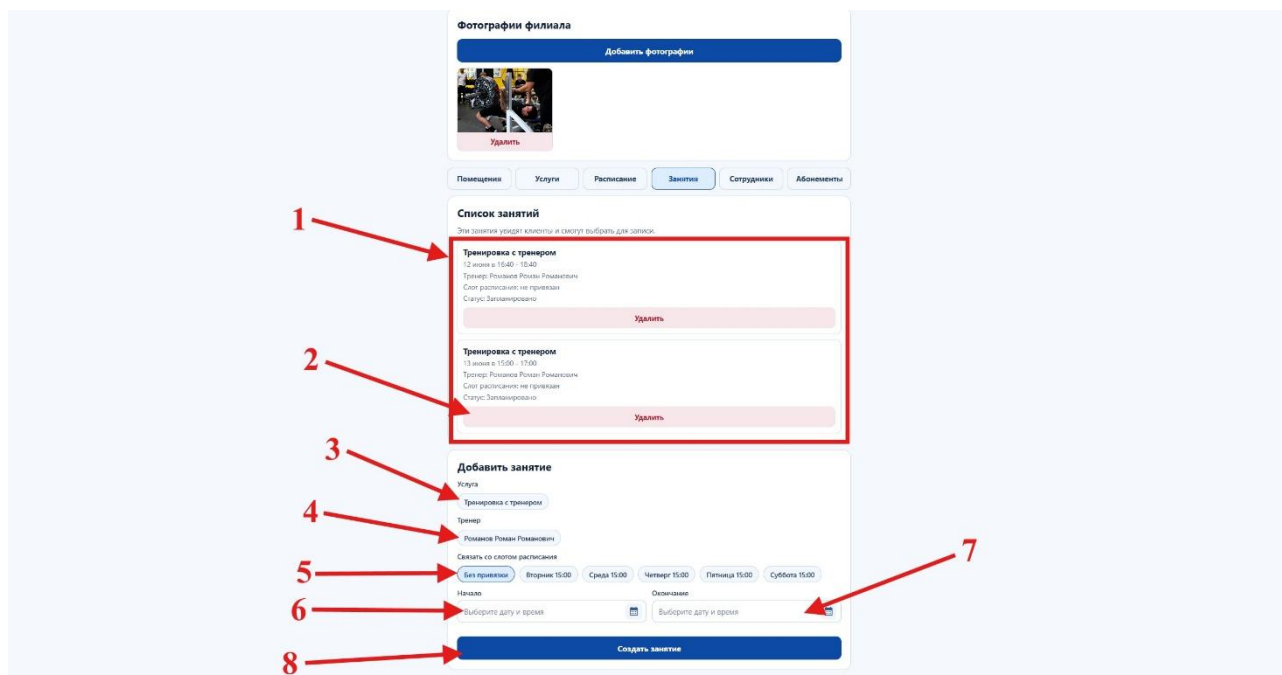


Рисунок 27 – Интерфейс управления тренировочными занятиями

Экран управления выбранным помещением показан на рисунке 28, в котором есть следующие элементы:

- 1 – ссылка возврата к филиалу;
- 2 – кнопка удаления помещения;
- 3 – кнопка добавления фотографий помещения;
- 4 – кнопка удаления фотографии помещения;
- 5 – поле для ввода названия оборудования;
- 6 – поле для ввода количества оборудования;
- 7 – кнопка добавления оборудования;
- 8 – кнопка открытия оборудования;
- 9 – кнопка удаления оборудования.

Экран помещения используется для учета залов, фотографий и оборудования внутри филиала. Благодаря такому интерфейсу администратор филиала всегда имеет актуальную информацию о каждом помещении: как оно выглядит (фотографии), что в нем находится (оборудование). Это упрощает планирование расписания, закупку инвентаря и информирование клиентов об условиях тренировок.

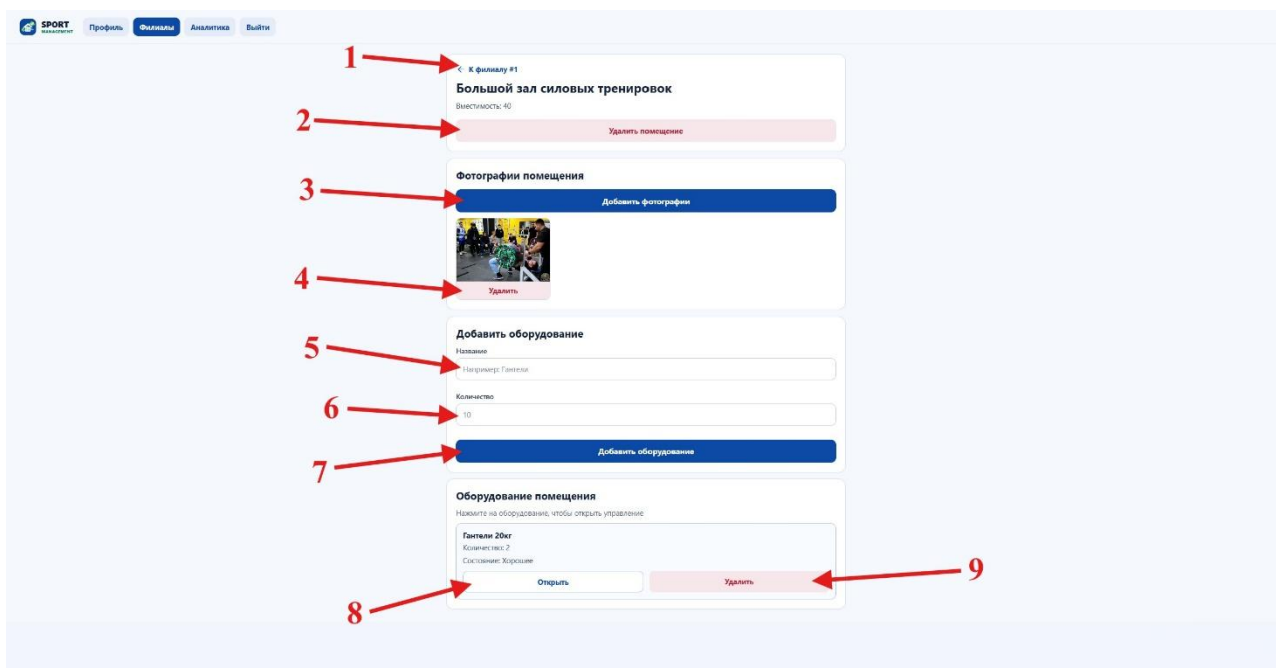


Рисунок 28 – Интерфейс управления помещением

Экран управления конкретной единицей оборудования приведен на рисунке 29, в котором обозначены:

- 1 – ссылка возврата к помещению;
- 2 – кнопка удаления оборудования;
- 3 – кнопка добавления фотографий оборудования;
- 4 – кнопка удаления фотографии оборудования.

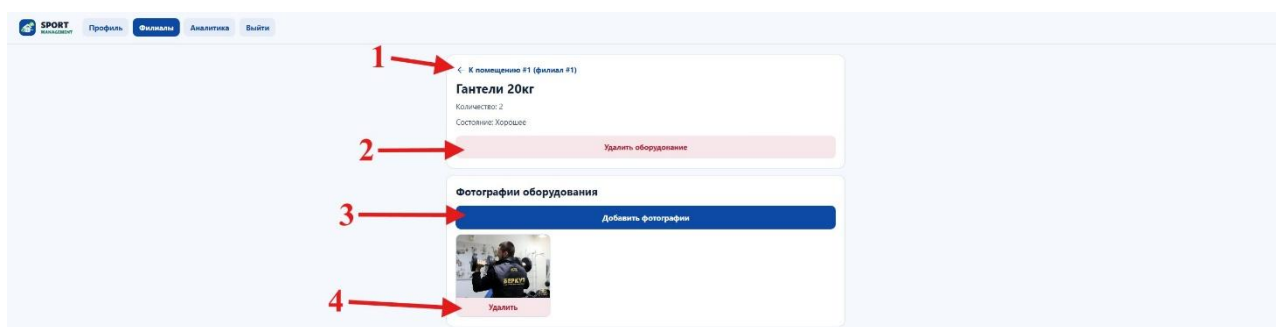


Рисунок 29 – Интерфейс управления оборудованием

При реализации интерфейса также были предусмотрены состояния загрузки, ошибок и пустых списков. Если данные еще не получены с сервера, пользователь видит сообщение о загрузке. Если список объектов пуст, приложение сообщает, что записи пока не добавлены. Для удаления важных

данных используются подтверждения, так как такие действия влияют на филиалы, помещения, сотрудников, услуги, расписание и абонементы.

Таким образом, интерфейс приложения обеспечивает выполнение основных пользовательских и административных сценариев. Пользователь может просматривать филиалы, оформлять абонементы и записываться на занятия, а администратор получает инструменты для управления структурой спортивной организации. Расшифровка элементов интерфейса по номерам показывает, что каждый экран имеет конкретное назначение и связан с общей логикой работы информационной системы.

4 Отладка и тестирование программного обеспечения

4.1 Выбор режима тестирования

Для разработанной информационной системы выбран комбинированный режим тестирования [35], включающий модульное, интеграционное и функциональное тестирование. Такой подход выбран с учетом архитектуры проекта: серверная часть реализована на ASP.NET Core с разделением на модули и слои, клиентская часть – на React Native, при этом значительная часть бизнес-логики сосредоточена на стороне сервера.

В проекте присутствует большое количество обработчиков команд и запросов, валидаторов и репозиториях, поэтому проверка только пользовательских сценариев не обеспечивает достаточного уровня надежности. По этой причине основа контроля качества построена на сочетании нескольких режимов тестирования, каждый из которых решает свою задачу.

Модульное тестирование выбрано для проверки отдельных компонентов в изоляции: обработчиков команд и запросов, валидаторов и вспомогательных методов. Такой режим позволяет выявлять ошибки в логике на раннем этапе и быстро локализовать источник проблемы.

Интеграционное тестирование используется для проверки взаимодействия между слоями приложения, в первую очередь между бизнес-логикой, репозиториями и базой данных. Для данного проекта это особенно важно, поскольку значимая часть функциональности связана с сохранением и извлечением данных, а также с проверкой ограничений на уровне доменной модели.

Функциональное тестирование применяется для контроля выполнения ключевых пользовательских сценариев. На этом уровне подтверждается, что система корректно обрабатывает основные действия пользователя: авторизацию, создание и просмотр сущностей, а также обработку типовых ошибок.

Выбор комбинированного режима тестирования обоснован тем, что каждый вид проверки закрывает отдельный класс рисков. Модульные тесты

контролируют корректность локальной логики, интеграционные – корректность взаимодействия компонентов, функциональные – соответствие системы требованиям на уровне реального использования.

4.2 Модульное тестирование

Модульное тестирование в проекте применялось для проверки корректности бизнес-логики на уровне отдельных компонентов без запуска всей системы целиком. Основной акцент был сделан на проверке обработчиков команд и запросов, а также валидаторов входных данных, так как именно в этих частях сосредоточены ключевые правила предметной области.

Тестовая подсистема реализована в отдельном проекте `SportManagementSystem.Tests`. Для написания модульных тестов использовались:

- `xUnit` – как основной фреймворк тестирования [36];
- `Moq` – для изоляции зависимостей и подмены репозиториях/сервисов [37];
- `Microsoft.NET.Test.Sdk` и `coverlet.collector` – для запуска тестов и контроля качества тестирования.

Проверка модулей выполнялась по принципу «один тест – один сценарий». Для каждого обработчика отдельно проверялись:

- корректный результат при валидных входных данных;
- обработка ошибок и пограничных условий;
- корректная реакция на отсутствие связанных сущностей;
- соблюдение бизнес-ограничений.

В рамках проекта реализованы 47 модульных тестов, покрывающих основные функциональные блоки: `Users`, `Assets`, `Clients`, `Scheduling`, `Services`, `Images`. Такой объем тестов позволяет контролировать стабильность логики при изменениях кода и снижает риск регрессии.

Перечень основных модульных тестов представлен в таблице 11.

Таблица 11 – Перечень модульных тестов

Тест	Описание	Ожидаемый результат	Актуальный результат
CreateBranchValidator_WhenRequestInvalid_ReturnsErrors	Валидатор создания филиала возвращает ошибки при некорректном запросе	Валидатор возвращает ошибки	Валидатор возвращает ошибки
CreateRoomValidator_WhenRequestInvalid_ReturnsErrors	Валидатор создания зала возвращает ошибки при некорректном запросе	Валидатор возвращает ошибки	Валидатор возвращает ошибки
CreateEquipmentValidator_WhenRequestInvalid_ReturnsErrors	Валидатор создания оборудования возвращает ошибки при некорректном запросе	Валидатор возвращает ошибки	Валидатор возвращает ошибки
CreateEquipmentValidator_WhenRequestValid_HasNoErrors	Валидатор создания оборудования не возвращает ошибок при корректном запросе	Ошибки валидации отсутствуют	Ошибки валидации отсутствуют
Handle_WhenRequestValid_CreatesBranch	Обработчик создает филиал при корректном запросе	Операция выполняется успешно	Операция выполняется успешно
Handle_WhenRoomMissing_ReturnsConflict	При отсутствии зала обработчик создания оборудования возвращает Conflict	Возвращается ошибка конфликта (Conflict)	Возвращается ошибка конфликта (Conflict)
Handle_WhenRequestValid_CreatesEquipment	Обработчик создает оборудование при корректном запросе	Операция выполняется успешно	Операция выполняется успешно
Handle_WhenBranchMissing_ReturnsConflict	При отсутствии филиала обработчик создания зала возвращает Conflict	Возвращается ошибка конфликта (Conflict)	Возвращается ошибка конфликта (Conflict)
Handle_WhenRequestValid_CreatesRoom	Обработчик создает зал при корректном запросе	Операция выполняется успешно	Операция выполняется успешно

По итогам (рисунок 30) выполнения модульного тестирования критических дефектов в изолированных компонентах не выявлено.

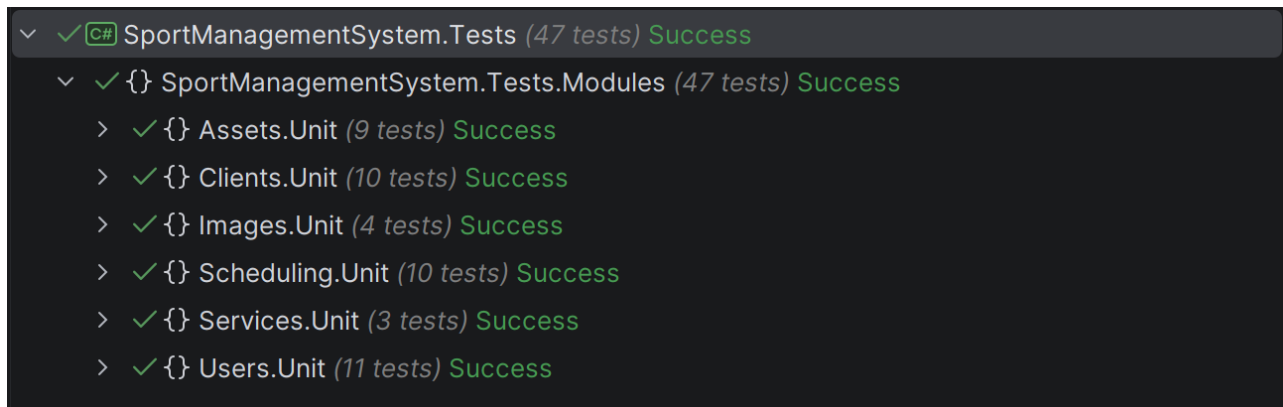


Рисунок 30 – Результат работы модульных тестов

Полученные результаты показывают, что обработчик и валидаторы корректно реализуют заявленные правила, а выбранный подход к unit-тестированию является достаточным для текущего этапа разработки.

4.3 Интеграционное тестирование

Интеграционное тестирование в проекте выполнялось для проверки совместной работы компонентов, которые в реальном режиме взаимодействуют между собой: репозитории, PostgreSQL и доменные сущности. В отличие от модульного тестирования, здесь проверялась не изолированная логика метода, а корректность работы на уровне связки инфраструктурных слоев.

Для запуска интеграционных сценариев использовались:

- xUnit – фреймворк тестирования;
- Testcontainers и Testcontainers.PostgreSql – подъем контейнера PostgreSQL для тестовой среды [38];
- Entity Framework Core (через реальные репозитории) – проверка операций с данными без моков.

Такой подход позволяет проверить, что CRUD операции действительно работают при взаимодействии с реальной базой данных, а не только в памяти.

В рамках проекта реализовано 15 интеграционных тестов. Перечень основных тестов приведен в таблице 12.

Таблица 12 – Перечень интеграционных тестов

Тест	Описание	Ожидаемый результат	Актуальный результат
BranchRepository_CreateAndGetById_Works	Проверка создания филиала в БД и последующего получения по идентификатору	Филиал сохраняется и корректно читается из БД	Филиал сохраняется и корректно читается из БД
RoomRepository_GetByBranch_ReturnsRooms	Проверка получения списка залов по идентификатору филиала	Возвращается список залов, связанных с указанным филиалом	Возвращается список залов, связанных с указанным филиалом
EquipmentRepository_CreateAndGetById_Works	Проверка создания оборудования и чтения по идентификатору	Оборудование успешно сохраняется и извлекается из БД	Оборудование успешно сохраняется и извлекается из БД
MembershipRepository_CreateAndGetById_Works	Проверка создания абонемента и получения записи по идентификатору	Абонемент успешно сохраняется и доступен для чтения	Абонемент успешно сохраняется и доступен для чтения
BookingRepository_CreateAndGetById_Works	Проверка создания бронирования и получения его по идентификатору	Бронирование корректно записывается и читается из БД	Бронирование корректно записывается и читается из БД
ImageRepository_CreateAndGetById_Works	Проверка создания записи изображения и получения по идентификатору	Запись изображения сохраняется и возвращается без потери данных	Запись изображения сохраняется и возвращается без потери данных
ImageRepository_Delete_RemovesEntity	Проверка удаления записи изображения из репозитория	После удаления запись отсутствует в БД	После удаления запись отсутствует в БД

Продолжение таблицы 12

Тест	Описание	Ожидаемый результат	Актуальный результат
ServiceScheduleRepository_CreateAndGetById_Works	Проверка создания расписания услуги и получения по идентификатору	Расписание услуги сохраняется и корректно извлекается	Расписание услуги сохраняется и корректно извлекается
TrainingSessionRepository_MarkDone_UpdatesStatus	Проверка изменения статуса тренировочной сессии на завершённый	Статус сессии обновляется и сохраняется в БД	Статус сессии обновляется и сохраняется в БД

По результатам (рисунок 31) интеграционного тестирования подтверждена корректная работа репозитория и согласованность взаимодействия между приложением и базой данных PostgreSQL.

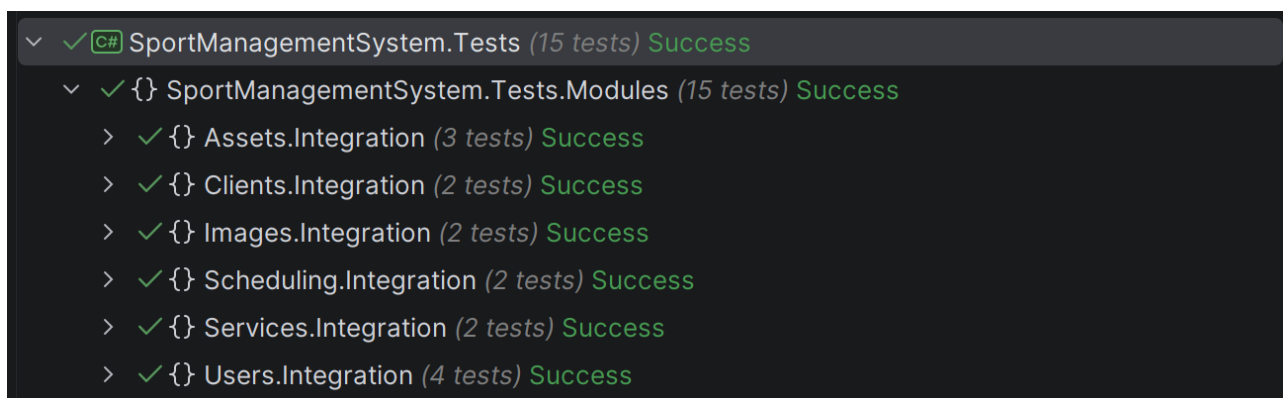


Рисунок 31 – Результат интеграционных тестов

Это снижает риск дефектов при переходе от unit-уровня к реальной эксплуатации системы.

4.4 Функциональное тестирование

Функциональное тестирование в проекте выполнялось для проверки соответствия системы заявленным пользовательским требованиям и сценариям использования. В отличие от модульных тестов, на данном уровне проверялась

работа системы «целиком»: от действий пользователя в интерфейсе до получения корректного результата от серверной части.

Сценарии функционального тестирования сформированы на основе ключевых прецедентов проекта:

- авторизация пользователя;
- регистрация клиента;
- создание бронирования;
- оформление абонемента;
- создание филиала администратором.

Проверка выполнялась на связке клиентского приложения и API с учетом ролевой модели и бизнес-ограничений.

Результаты функционального тестирования представлены в таблице 13.

Таблица 13 – Результаты функционального тестирования

Тест	Описание	Ожидаемый результат	Актуальный результат
Авторизация корректные данные	Проверка входа пользователя с валидными учетными данными	Пользователь авторизуется, получает токен, открываются разделы согласно роли	Пользователь авторизуется, получает токен, открываются разделы согласно роли
Авторизация неверный пароль	Проверка попытки входа с неверным паролем	Система отклоняет вход и возвращает ошибку авторизации	Система отклоняет вход и возвращает ошибку авторизации
Регистрация клиента валидные данные	Проверка регистрации нового клиента с корректно заполненной формой	Создается учетная запись клиента, отображается успешный результат регистрации	Создается учетная запись клиента, отображается успешный результат регистрации

Продолжение таблицы 13

Тест	Описание	Ожидаемый результат	Актуальный результат
Регистрация дублирующий email	Проверка регистрации клиента с email, уже существующим в системе	Регистрация отклоняется, отображается сообщение о конфликте уникальности	Регистрация отклоняется, отображается сообщение о конфликте уникальности
Создание бронирования валидный сценарий	Проверка оформления бронирования при наличии доступного слота	Бронирование создается	Бронирование создается
Создание бронирования недоступный слот	Проверка оформления бронирования при некорректных/недоступных данных слота	Система не создает запись и возвращает сообщение об ошибке	Система не создает запись и возвращает сообщение об ошибке
Оформление абонемента валидный сценарий	Проверка создания абонемента для существующей спортивной услуги	Абонемент создается, присваивается статус, данные доступны для просмотра	Абонемент создается, присваивается статус, данные доступны для просмотра
Оформление абонемента услуга отсутствует	Проверка создания абонемента для несуществующей услуги	Система возвращает ошибку, абонемент не создается	Система возвращает ошибку, абонемент не создается
Создание филиала admin роль	Проверка создания филиала пользователем с ролью администратора	Филиал успешно создается и отображается в общем списке филиалов	Филиал успешно создается и отображается в общем списке филиалов
Создание филиала без прав	Проверка создания филиала пользователем без роли администратора	Операция отклоняется, доступ запрещен	Операция отклоняется, доступ запрещен

По результатам функционального тестирования подтверждено, что ключевые пользовательские сценарии реализованы корректно и соответствуют

требованиям проекта. Проверка позитивных и негативных сценариев показала устойчивое поведение системы как при стандартной работе, так и при ошибочных действиях пользователя.

5 Контроль качества программного обеспечения

5.1 Выбор метрик программного кода

Оценка качества программного кода требует не одного показателя, а набора взаимодополняющих метрик. Одна часть метрик должна описывать объем и внутреннюю сложность реализации, другая – отражать особенность работы с данными, третья – показывать состояние архитектуры. Дополнительно нужен автоматизированный интегральный индикатор, пригодный для регулярного контроля.

Метрика SLOC (Source Lines of Code) выбрана как базовая количественная характеристика кода. Она фиксирует размер программной системы в строках, позволяет сравнивать модули по масштабу и служит опорным параметром при анализе других метрик. Для практической оценки трудоемкости сопровождения значение SLOC является необходимым исходным ориентиром.

Группа метрик Холстеда используется для формального анализа структуры исходного текста через операторы и операнды. На ее основе определяются словарь, длина, объем и вычислительная сложность программы. Такие показатели полезны при поиске перегруженных участков кода и позволяют количественно оценивать, насколько трудно читать и изменять конкретные фрагменты.

Метрика Чепина применяется для анализа ролей переменных в программе (Р, М, С, Т). Этот подход дает возможность оценивать качество логики обработки данных: какие переменные действительно участвуют в вычислениях, какие управляют потоком выполнения и какие создают лишнюю информационную нагрузку. В результате повышается объективность оценки понятности модулей.

Для объектного уровня используются метрики Р. Мартина: Са, Се, I, А и D. Данные показатели характеризуют зависимости между компонентами, степень устойчивости модулей и соответствие архитектуры принципам баланса между абстракцией и стабильностью. Такой выбор оправдан тем, что при

развитии проекта именно архитектурная связанность чаще всего определяет стоимость изменений.

В качестве автоматизированного индикатора принят Maintainability Index (MI) [39]. Он рассчитывается средствами статического анализа и представляет единую числовую оценку сопровождаемости кода. Преимущество MI состоит в том, что метрика подходит для периодического мониторинга и позволяет отслеживать динамику качества без субъективных интерпретаций.

Выбранный набор метрик охватывает ключевые аспекты качества программного кода: объем, структурную и алгоритмическую сложность, работу с переменными, архитектурную устойчивость и итоговую сопровождаемость.

5.2 Оценка качества программного кода

5.2.1 Метрика SLOC

На этапе контроля качества для моего проекта была рассчитана метрика SLOC, которая позволяет количественно оценить объем исходного кода. Расчет выполнялся по двум частям системы: серверной и клиентской.

Чтобы результат отражал именно разработанный прикладной код, из анализа были исключены служебные и автогенерируемые файлы (bin/obj, dist), а также миграции и тестовые файлы.

Результат расчета метрики SLOC представлен в таблице 14.

Таблица 14 – Результат расчета метрики SLOC

Показатель	Значение	Примечание
Количество пустых строк	1497	Строка без кода и комментариев
Количество строк с комментариями	2784	Включая однострочные и блочные комментарии
Общее количество строк	12237	Полный объем анализируемых файлов

Продолжение таблицы 14

Показатель	Значение	Примечание
Процент комментариев	22,7507%	Доля строк с комментариями в общем объеме
Количество модулей (файлов)	246	Сервер и клиент
Среднее число строк на модуль	49,7439	Общее количество строк, деленное на число модулей
Среднее число строк исходного кода на модуль	33,0407	Код без пустых строк и комментариев в среднем на модуль
SLOC (строки кода без пустых и комментариев)	8128	Итоговый показатель

По полученным данным можно сделать вывод, что кодовая база проекта имеет умеренный объем: итоговый SLOC составил 8128 строк. Доля комментариев (22,75%) показывает, что код в достаточной степени документирован. Среднее число строк на модуль (49,74) и среднее число строк чистого кода на модуль (33,04) позволяют считать структуру модулей компактной и удобной для чтения. В целом результаты метрики SLOC подтверждают, что текущая реализация подходит для дальнейшего сопровождения и развития системы.

5.2.2 Метрика Холстеда

Для более детальной оценки сложности кода в проекте дополнительно была рассчитана метрика Холстеда. В отличие от SLOC, она учитывает не только объем, но и структуру исходного текста через операторы и операнды. Расчет выполнялся по серверной и клиентской частям проекта и с теми же правилами отбора файлов, что и для метрики SLOC.

Для вычисления использовались базовые параметры:

- число уникальных операторов, обозначается $n1$;

- число уникальных операндов, обозначается n_2 ;
- общее число операторов, обозначается N_1 ;
- общее число операндов, обозначается N_2 .

Далее рассчитывались произвольные показатели:

- словарь программы, обозначается n ;
- длина программы, обозначается N ;
- теоретическая длина, обозначается N' ;
- объем программы, обозначается V ;
- сложность программы, обозначается D .

Результат расчета метрики Холстеда представлены в таблице 15.

Таблица 15 – Результат расчета метрики Холстеда

Параметр	Значение
n_1	33
n_2	2293
N_1	29053
N_2	19035
n	2326
N	48088
N'	26130,7683
V	537680,6576
D	136,9474

Полученные значения показывают, что код проекта имеет достаточно большой словарь и значительный общий объем операций, что ожидаемо для системы с серверной и клиентской частями. Наибольший вклад в итоговые показатели вносит количество операндов, то есть разнообразие используемых сущностей, полей, параметров и переменных. По результатам метрики Холстеда можно сделать вывод, что проект относится к коду средней и выше средней

структурной сложности, поэтому при дальнейшем развитии системы важно сохранить модульность и единый стиль реализации.

5.2.3 Метрика Чепина

Для оценки качества программной логики в проекте была рассчитана метрика Чепина. Данная метрика позволяет проанализировать, как в коде используются переменные, и за счет этого оценить понятность и сложность модулей.

В рамках расчета все переменные распределяются по четырем группам:

- переменные, используемые как хранение/передача данных, обозначаемые Р;
- переменные, которые изменяются в ходе выполнения программы, обозначаемые М;
- переменные, управляющие ветвлением и логикой выполнения, обозначаемые С;
- неиспользуемые или малозначимые переменные, обозначаемые Т.

Полная метрика Чепина рассчитывается по формуле:

$$Q = a1 \times P \times a2 \times M + a3 \times C + a4 \times T,$$

где а1 – стандартные весовой коэффициент равный 1;

а2 – стандартные весовой коэффициент равный 2;

а3 – стандартные весовой коэффициент равный 3;

а4 – стандартные весовой коэффициент равный 0,5.

Результаты расчета метрики Чепина представлены в таблице 16.

Таблица 16 – Результаты расчета метрики Чепина

Параметр	Значение
Р	95
М	237
С	158
Т	68

Продолжение таблицы 16

Параметр	Значение
a1	1
a2	2
a3	3
a4	0,5
Q	1077

Полученное значение метрики Чепина показывает, что основную сложность в проекте формируют группы М и С, то есть изменяемые и управляющие переменные. Это характерно для прикладной системы с выраженной бизнес-логикой и большим количеством условий. При дальнейшем развитии проекта целесообразно уделять внимание сокращению управляющих зависимостей и контролю за переменными Т, чтобы поддерживать код в удобном для сопровождения состоянии.

5.2.4 Метрика Р. Мартина

Для оценки качества архитектуры проекта использованы объектные метрики Р. Мартина. В отличие от метрик уровня строк и операторов, они показывают устойчивость и связность модулей на уровне зависимостей между классами и пространствами имен.

Расчет выполнен для серверной части проекта, где в качестве анализируемой категории рассматривались классы доменной и прикладной логики (SportManagementSystem.Modules).

В рамках расчета определялись следующие показатели:

- Ca (Afferent Coupling) – количество внешних классов, зависящих от классов категории;
- Ce (Efferent Coupling) – количество классов категории, зависящих от внешних классов;

- I (Instability) – нестабильность;
- A (Abstractness) – абстрактность;
- D (Distance from Main Sequence) – расстояние до главной последовательности.

Результат расчета метрики Р. Мартина представлен в таблице 17.

Таблица 17 – Результат расчета метрики Р. Мартина

Параметр	Значение
Ca	17
Ce	57
nA	14
nAll	121
I	0,7703
A	0,1157
D	0,1140

По полученным результатам категория имеет повышенную нестабильность ($I = 0,77$), то есть сильнее зависит от внешних компонентов, чем влияет на них. Значение абстрактности составляет 0,12, что указывает на наличие абстракций, но их доля умеренная. Расстояние до главной последовательности 0,11 показывает, что архитектура находится в относительно сбалансированном состоянии.

5.2.5 Индикатор Maintainability Index

В качестве автоматизированной метрики качества кода для проекта использован индекс поддерживаемости Maintainability Index (MI). Этот показатель позволяет в одном числовом значении оценить, насколько удобно сопровождать и развивать программный код.

Расчет выполнялся на основе стандартной формулы:

$$MI = \max(0, (171 - 5,2 \times \ln V - 0,23 \times CC - 16,2 \times \ln LOC) \times \frac{100}{171}),$$

где V – объем кода по Холстеду;

СС – цикломатическая сложность;

LOC – количество строк кода без пустых строк и комментариев.

Результат расчета индикатора Maintainability Index представлен в таблице 18.

Таблица 18 – Результат расчета индикатора Maintainability Index

Параметр	Значение
V	2185,6937
LOC	33,0407
СС	4,2214
MI	42,9132

Полученное значение 42,91 показывает, что код проекта находится в зоне средней сопровождаемости. Это означает, что текущая реализация в целом пригодна для дальнейшей доработки, но отдельные участки требуют упрощения. Для повышения индекса поддерживаемости в первую очередь целесообразно снижать сложность методов, уменьшать связанность между модулями и сохранять единый стиль проектирования кода.

6 Планирование разработки и оценка бюджета

При разработке проекта было выполнено планирование работ с разбивкой на этапы, определением сроков, трудоемкости и стоимости. В качестве базового инструмента планирования выбрана диаграмма Ганта (рисунок 32), так как она наглядно показывает последовательность задач, их длительность и взаимосвязи.

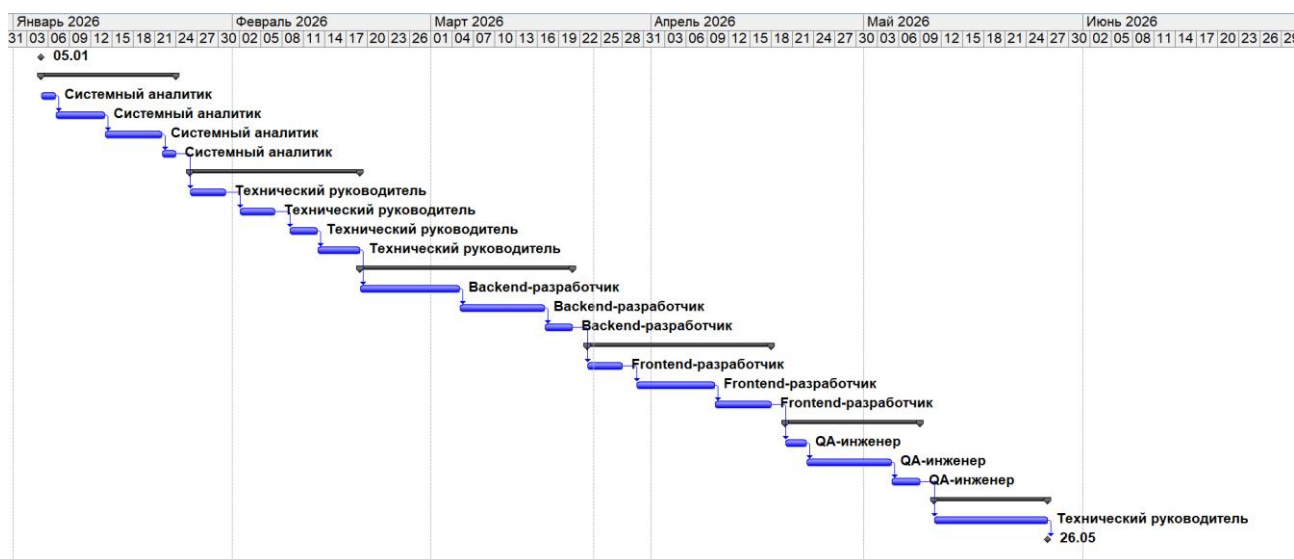


Рисунок 32 – Диаграмма Ганта

Для расчета бюджета использовался ресурсный подход: стоимость каждого этапа определялась через трудозатраты и часовую ставку исполнителя. Для оценки бюджета была принята модель проектной команды с распределением ролей и частичной занятостью участников (рисунок 33).

Название ресурса	Тип	Единицы измерения материалов	Краткое название	Группа	Макс. единиц	Стандартная ставка
Системный аналитик	Трудовой		C		100%	650,00 Р/ч
Backend-разработчик	Трудовой		B		100%	850,00 Р/ч
Frontend-разработчик	Трудовой		F		100%	750,00 Р/ч
QA-инженер	Трудовой		Q		100%	500,00 Р/ч
Технический руководитель	Трудовой		T		100%	900,00 Р/ч

Рисунок 33 – Ресурсы проекта

План разработки был сформирован по основным этапам жизненного цикла проекта. Структура этапов и подзадач календарного плана показана на рисунке 34.

Название задачи	Длительность	Начало	Окончание	Предшественники	Названия ресурсов
Начало проекта	0 дней	Пн 05.01.26	Пн 05.01.26		
Постановка задачи и анализ	15 дней	Пн 05.01.26	Пт 23.01.26		
Формирование целей	2 дней	Пн 05.01.26	Вт 06.01.26		Системный аналитик
Анализ предметной области	5 дней	Ср 07.01.26	Вт 13.01.26	3	Системный аналитик
Подготовка use-case и диаграмм	6 дней	Ср 14.01.26	Ср 21.01.26	4	Системный аналитик
Планирование работ и ресурсов	2 дней	Чт 22.01.26	Пт 23.01.26	5	Системный аналитик
Проектирование решения	18 дней	Пн 26.01.26	Ср 18.02.26		
Проектирование архитектуры	5 дней	Пн 26.01.26	Пт 30.01.26	6	Технический руководитель
Проектирование структуры данных	5 дней	Пн 02.02.26	Пт 06.02.26	8	Технический руководитель
Проектирование API-контрактов	4 дней	Пн 09.02.26	Чт 12.02.26	9	Технический руководитель
Проектирование пользовательского интерфейса	4 дней	Пт 13.02.26	Ср 18.02.26	10	Технический руководитель
Реализация серверной части	22 дней	Чт 19.02.26	Пт 20.03.26		
Реализация модулей данных	10 дней	Чт 19.02.26	Ср 04.03.26	11	Backend-разработчик
Реализация API, авторизация	8 дней	Чт 05.03.26	Пн 16.03.26	13	Backend-разработчик
Интеграция с хранилищем	4 дней	Вт 17.03.26	Пт 20.03.26	14	Backend-разработчик
Реализация клиентской части	20 дней	Пн 23.03.26	Пт 17.04.26		
Реализация экранов администрирования	5 дней	Пн 23.03.26	Пт 27.03.26	15	Frontend-разработчик
Реализация экранов пользователя	9 дней	Пн 30.03.26	Чт 09.04.26	17	Frontend-разработчик
Интеграция клиента с сервером	6 дней	Пт 10.04.26	Пт 17.04.26	18	Frontend-разработчик
Тестирование и стабилизация	15 дней	Пн 20.04.26	Пт 08.05.26		
Подготовка тест-плана	3 дней	Пн 20.04.26	Ср 22.04.26	19	QA-инженер
Функциональное, модульное тестирование	8 дней	Чт 23.04.26	Пн 04.05.26	21	QA-инженер
Исправление дефектов	4 дней	Вт 05.05.26	Пт 08.05.26	22	QA-инженер
Оформление отчетных материалов	12 дней	Пн 11.05.26	Вт 26.05.26		
Подготовка пояснительной записки	12 дней	Пн 11.05.26	Вт 26.05.26	23	Технический руководитель
Конец проекта	0 дней	Вт 26.05.26	Вт 26.05.26	25	

Рисунок 34 – Календарный план разработки

На основе календарного плана была выполнена оценка трудозатрат и стоимости проекта, результат которой приведен на рисунке 35.

	Начало	Окончание
Текущее	Пн 05.01.26	Вт 26.05.26
Базовое	НД	НД
Фактическое	НД	НД
Отклонение	0д	0д

	Длительность	Трудозатраты	Затраты
Текущие	102д	816ч	623 600,00 Р
Базовые	0д?	0ч	0,00 Р
Фактические	0д	0ч	0,00 Р
Оставшиеся	102д	816ч	623 600,00 Р

Рисунок 35 – Оценка стоимости

По результатам планирования общая трудоемкость разработки составила 816 человеко-часов (102 рабочих дня), а итоговая оценка бюджета 623 600 рублей.

Применимый подход к планированию позволил заранее определить структуру работ, распределить время по этапам и получить обоснованную оценку затрат на разработку проекта.

Заключение

В ходе выполнения дипломной работы была разработана информационная система управления спортивной организацией. Основная цель работы заключалась в создании программного решения, которое позволяет автоматизировать ключевые процессы спортивного клуба или фитнес-центра: работу с пользователями, филиалами, услугами, расписанием, абонементом и записями клиентов на занятия. В результате была создана клиент-серверная система, объединяющая пользовательские и административные функции.

Перед реализацией проекта был выполнен анализ предметной области и существующих программных решений, применяемых для управления спортивными организациями. Это позволило определить основные требования к системе и выделить наиболее важные сценарии работы. Для клиента такими сценариями стали просмотр филиалов, выбор услуги, оформление абонемента и запись на занятие. Для администратора и управляющего важными стали управление филиалами, помещениями, оборудованием, услугами, расписанием, сотрудниками и шаблонами абонементов.

В процессе разработки была выбрана клиент-серверная архитектура. Серверная часть реализована на ASP.NET Core и отвечает за обработку запросов, бизнес-логику, авторизацию, проверку прав доступа и работу с данными. Клиентская часть реализована на React Native с использованием TypeScript и обеспечивает взаимодействие пользователя с системой через удобный интерфейс. Такой подход позволил разделить прикладную логику и пользовательское представление, что упрощает сопровождение и дальнейшее развитие проекта.

Для хранения данных была разработана структура базы данных на PostgreSQL. В ней размещаются основные сведения о пользователях, филиалах, помещениях, оборудовании, услугах, расписании, занятиях, абонементе и записях клиентов. Для хранения изображений используется MinIO, а для аналитических событий предусмотрено хранилище ClickHouse. Такое

разделение позволяет использовать разные хранилища по назначению и делает систему более гибкой.

В реализованном приложении пользователь может зарегистрироваться, войти в систему, просмотреть филиалы спортивной организации, изучить доступные услуги, оформить абонемент и записаться на занятие. Администратор или управляющий получает возможность настраивать структуру филиалов, добавлять помещения и оборудование, управлять услугами, расписанием, тренировочными занятиями, сотрудниками и абонементом. Интерфейс построен таким образом, чтобы основные действия были доступны из понятных разделов, а данные отображались в виде списков, карточек и форм.

Работоспособность системы была проверена с помощью модульного, интеграционного и функционального тестирования. Модульные тесты использовались для проверки отдельных элементов бизнес-логики, интеграционные тесты подтвердили корректную работу приложения с базой данных, а функциональное тестирование показало выполнение основных пользовательских сценариев. По результатам проверки критических ошибок в ключевых сценариях выявлено не было.

Также была выполнена оценка качества программного кода и планирование разработки. Для анализа применялись метрики SLOC, Холстеда, Чепина, метрики Р. Мартина и Maintainability Index. Полученные результаты показали, что проект имеет средний уровень сложности и может сопровождаться при сохранении модульной структуры. Оценка трудоемкости и бюджета позволила определить примерные затраты на разработку системы и рассмотреть проект не только с технической, но и с организационной точки зрения.

Практическая значимость работы заключается в том, что разработанная система может быть использована как основа для внедрения в спортивной организации. Она позволяет хранить данные в едином месте, снижает количество ручных операций и упрощает взаимодействие между клиентами и сотрудниками. В дальнейшем систему можно расширить за счет онлайн-оплаты,

уведомлений, более подробной аналитики, отчетов по посещаемости и выручке, а также полноценной мобильной версии приложения.

Таким образом, поставленная цель дипломной работы была достигнута. В результате была разработана информационная система, которая решает основные задачи автоматизации спортивной организации, включает серверную и клиентскую части, базу данных, пользовательский интерфейс и механизмы проверки работоспособности. Полученное решение может применяться как основа для дальнейшего развития и адаптации под реальные условия работы спортивного клуба.

Список использованных источников

1. Соммервилл И. Инженерия программного обеспечения. 6-е изд. – М.: Вильямс, 2002. – 624 с.
2. Прессман Р. Инженерия программного обеспечения: практический подход. – М.: Вильямс, 2005. – 704 с.
3. Орлов С. А. Технологии разработки программного обеспечения. – СПб.: Питер, 2012. – 608 с.
4. ISO/IEC 25010:2011. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программного обеспечения. – Geneva: International Organization for Standardization, 2011.
5. Gymdesk. Программное обеспечение для управления спортивным залом [Электронный ресурс]. – URL: <https://gymdesk.com/> (дата обращения: 19.05.2026).
6. PushPress. Программное обеспечение для управления спортивным залом [Электронный ресурс]. – URL: <https://www.pushpress.com/> (дата обращения: 20.05.2026).
7. GymMaster. Программное обеспечение для управления спортивным залом [Электронный ресурс]. – URL: <https://www.gymmaster.com/> (дата обращения: 21.05.2026).
8. Басс Л., Клементс П., Казман Р. Архитектура программного обеспечения на практике. 3-е изд. – СПб.: Питер, 2013. – 640 с.
9. Фаулер М. Архитектура корпоративных программных приложений. – М.: Вильямс, 2006. – 544 с.
10. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. – 352 с.
11. Эванс Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. – М.: Вильямс, 2011. – 448 с.

12. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2020. – 368 с.
13. MinIO. Документация MinIO [Электронный ресурс]. – URL: <https://docs.min.io/> (дата обращения: 24.04.2026).
14. PostgreSQL Global Development Group. Документация PostgreSQL [Электронный ресурс]. – URL: <https://www.postgresql.org/docs/> (дата обращения: 20.04.2026).
15. Meta Platforms. Документация React Native [Электронный ресурс]. – URL: <https://reactnative.dev/docs/getting-started> (дата обращения: 30.04.2026).
16. Docker Inc. Документация Docker [Электронный ресурс]. – URL: <https://docs.docker.com/> (дата обращения: 26.04.2026).
17. Docker Inc. Документация Docker Compose [Электронный ресурс]. – URL: <https://docs.docker.com/compose/> (дата обращения: 28.04.2026).
18. Коннолли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е изд. – М.: Вильямс, 2003. – 1440 с.
19. Дейт К. Дж. Введение в системы баз данных. 8-е изд. – М.: Вильямс, 2005. – 1328 с.
20. Microsoft. Документация Entity Framework Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/en-us/ef/core/> (дата обращения: 16.04.2026).
21. Npgsql. Документация провайдера Npgsql Entity Framework Core [Электронный ресурс]. – URL: <https://www.npgsql.org/efcore/> (дата обращения: 18.04.2026).
22. ClickHouse. Документация ClickHouse [Электронный ресурс]. – URL: <https://clickhouse.com/docs/en> (дата обращения: 22.04.2026).
23. Microsoft. Документация ASP.NET Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/en-us/aspnet/core/> (дата обращения: 14.04.2026).
24. Microsoft. Документация .NET [Электронный ресурс]. – URL: <https://learn.microsoft.com/en-us/dotnet/> (дата обращения: 12.04.2026).

25. Филдинг Р. Т. Архитектурные стили и проектирование сетевых программных архитектур [Электронный ресурс]: диссертация. – University of California, Irvine, 2000. – URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (дата обращения: 08.04.2026).
26. Мартин Р. Чистый код. Создание, анализ и рефакторинг. – СПб.: Питер, 2019. – 464 с.
27. FluentValidation. Документация FluentValidation [Электронный ресурс]. – URL: <https://docs.fluentvalidation.net/> (дата обращения: 14.05.2026).
28. Джонс М., Брэдли Дж., Сакимура Н. RFC 7519: JSON Web Token (JWT) [Электронный ресурс]. – IETF, 2015. – URL: <https://datatracker.ietf.org/doc/html/rfc7519> (дата обращения: 10.04.2026).
29. Hangfire. Документация Hangfire [Электронный ресурс]. – URL: <https://docs.hangfire.io/> (дата обращения: 06.05.2026).
30. Swagger. Спецификация OpenAPI [Электронный ресурс]. – URL: <https://swagger.io/specification/> (дата обращения: 16.05.2026).
31. Lygin04. SportManagementSystem [Электронный ресурс]: исходный код серверной части информационной системы. – GitHub. – URL: <https://github.com/Lygin04/SportManagementSystem> (дата обращения: 23.05.2026).
32. Microsoft. Документация TypeScript [Электронный ресурс]. – URL: <https://www.typescriptlang.org/docs/> (дата обращения: 02.05.2026).
33. Yandex. JavaScript API Яндекс Карт [Электронный ресурс]. – URL: <https://yandex.com/dev/maps/jsapi/> (дата обращения: 04.05.2026).
34. Lygin04. SportManagementClient [Электронный ресурс]: исходный код клиентской части информационной системы. – GitHub. – URL: <https://github.com/Lygin04/SportManagementClient> (дата обращения: 25.05.2026).
35. Майерс Г., Баджетт Т., Сандлер К. Искусство тестирования программ. 3-е изд. – М.: Диалектика, 2012. – 272 с.
36. xUnit.net. Документация xUnit.net [Электронный ресурс]. – URL: <https://xunit.net/> (дата обращения: 08.05.2026).

37. Моq. Документация Моq [Электронный ресурс]. – URL: <https://documentation.help/Моq/> (дата обращения: 10.05.2026).
38. Testcontainers. Руководство по Testcontainers for .NET [Электронный ресурс]. – URL: <https://testcontainers.com/guides/getting-started-with-testcontainers-for-dotnet/> (дата обращения: 12.05.2026).
39. Холстед М. Х. Начала науки о программах. – М.: Финансы и статистика, 1981. – 128 с.

Приложение А
(справочное)
Глоссарий

Абонемент – запись о приобретенном клиентом праве пользоваться определенной услугой в течение заданного периода или в пределах установленного лимита посещений.

Администратор филиала – пользователь системы, который управляет данными конкретного филиала: помещениями, оборудованием, услугами, расписанием, сотрудниками и абонементом.

Аналитика – раздел системы, предназначенный для просмотра показателей активности клиентов, просмотров филиалов, записей на занятия и покупок абонементов.

Аналитическое событие – запись о действии пользователя, например просмотре карточки филиала, открытии подробной информации, покупке абонемента или записи на занятие.

API – программный интерфейс, через который клиентская часть обменивается данными с серверной частью приложения.

ASP.NET Core – платформа разработки серверной части приложения, используемая для создания REST API, обработки запросов и выполнения бизнес-логики.

Аутентификация – процесс проверки личности пользователя по email и паролю перед предоставлением доступа к системе.

Авторизация – проверка прав пользователя после входа в систему, определяющая доступность функций в зависимости от роли.

База данных – организованное хранилище данных, используемое для сохранения пользователей, филиалов, услуг, расписания, абонементов, записей и других сущностей системы.

Бронирование – запись клиента на конкретное тренировочное занятие с фиксацией статуса и времени создания записи.

Валидатор – компонент, проверяющий корректность входных данных перед выполнением операции.

Внешний ключ – поле таблицы базы данных, которое связывает запись с записью другой таблицы и поддерживает ссылочную целостность.

Доменная сущность – объект предметной области, отражающий важную для системы сущность, например филиал, клиент, услугу или абонемент.

JWT – токен, используемый для передачи сведений об авторизованном пользователе и его роли между клиентом и сервером.

Запись на занятие – клиентская операция, при которой пользователь выбирает тренировочную сессию и подтверждает участие в ней.

Изображение – медиафайл, связанный с филиалом, помещением, оборудованием или профилем пользователя.

Индекс поддерживаемости – метрика качества кода, показывающая, насколько удобно сопровождать и развивать программный код.

Интеграционное тестирование – проверка совместной работы компонентов системы, например репозиториев, базы данных и доменных сущностей.

Клиент – пользователь системы, который просматривает филиалы, оформляет абонементы и записывается на занятия.

Клиентская часть – часть приложения, отвечающая за пользовательский интерфейс и взаимодействие пользователя с системой.

Концептуальная модель базы данных – обобщенное описание основных сущностей предметной области и связей между ними без детализации физических таблиц.

Логическая модель базы данных – формальное описание таблиц, атрибутов, ключей и связей, используемое перед физической реализацией базы данных.

Миграция базы данных – механизм изменения структуры базы данных при развитии модели данных приложения.

Модульное тестирование – проверка отдельных компонентов программы в изоляции от остальной системы.

Модульная архитектура – подход к организации приложения, при котором функциональность разделяется на самостоятельные предметные модули.

Объектное хранилище – хранилище файлов, используемое для размещения изображений отдельно от реляционной базы данных.

Помещение – зал или другая зона филиала, где могут размещаться оборудование и проводиться услуги.

Пользовательская роль – категория пользователя, определяющая его права в системе, например клиент, сотрудник, администратор или управляющий.

Предметная область – область деятельности, для которой создается система; в данной работе это управление спортивной организацией.

Расписание – набор временных слотов, определяющих, когда, где и какая услуга может проводиться.

Реляционная база данных – база данных, в которой информация хранится в таблицах, связанных между собой ключами.

Репозиторий – компонент серверной части, отвечающий за получение и сохранение данных в базе данных.

Серверная часть – часть приложения, выполняющая бизнес-логику, обработку запросов, проверку прав доступа и работу с данными.

Сотрудник – пользователь, связанный с филиалом и выполняющий административные или рабочие функции, например менеджер или тренер.

Спортивная услуга – услуга филиала, которую может получить клиент, например тренировка, фитнес, массаж или сауна.

Тренировочная сессия – конкретное занятие с датой, временем, тренером, услугой и статусом проведения.

Физическая модель базы данных – описание фактической структуры хранения данных с таблицами, типами полей, первичными и внешними ключами.

Филиал – отдельный спортивный объект организации с адресом, координатами, помещениями, услугами и сотрудниками.

Шаблон абонеента – предварительно настроенный вариант абонеента, содержащий услугу, цену, срок действия и лимит посещений.

ClickHouse – аналитическая СУБД, используемая в проекте для хранения событий активности клиентов и построения отчетов.

CQRS – подход, при котором операции изменения данных и операции чтения данных разделяются на команды и запросы.

Docker Compose – инструмент для запуска нескольких контейнеров приложения, например API, PostgreSQL, MinIO и ClickHouse.

Entity Framework Core – ORM-инструмент, используемый для работы серверной части с базой данных PostgreSQL через C#-классы.

Feature-ориентированная структура – организация клиентского приложения по функциональным областям, например авторизация, филиалы, клиентские операции и профиль.

Hangfire – инструмент для выполнения фоновых задач в серверной части приложения.

MediatR – библиотека, используемая для передачи команд и запросов от контроллеров к обработчикам бизнес-логики.

MinIO – объектное хранилище, применяемое для размещения изображений филиалов, помещений, оборудования и аватаров.

PostgreSQL – основная реляционная СУБД проекта, используемая для хранения бизнес-данных системы.

React Native – технология разработки клиентской части, позволяющая использовать единую кодовую базу для web- и мобильных платформ.

REST API – архитектурный подход к построению API, при котором клиентская часть обращается к серверу через HTTP-запросы.

Swagger – инструмент для описания и просмотра API серверной части приложения.

Testcontainers – инструмент, позволяющий запускать контейнеры баз данных и других сервисов при интеграционном тестировании.

TypeScript – язык программирования, расширяющий JavaScript статической типизацией и используемый в клиентской части проекта.

xUnit – фреймворк тестирования, используемый для модульных и интеграционных тестов серверной части.

Приложение Б
(обязательное)
Листинг серверной части программы

Листинг Б.1 – Файл Program.cs

```

using Hangfire;
using Microsoft.EntityFrameworkCore;
using SportManagementSystem.Data;
using SportManagementSystem.BuildingBlocks.Time;
using SportManagementSystem.Extensions;
using SportManagementSystem.Middleware;
using
SportManagementSystem.Modules.Scheduling.Application.Dispatchers;

var builder = WebApplication.CreateBuilder(args);
var configuration = builder.Configuration;

builder.Services.AddRouting(options => options.LowercaseUrls =
true);
builder.Services.Configure<AppTimeOptions>(configuration.GetSection(
"Time"));
builder.Services.AddSingleton<IAppClock, SystemAppClock>();

builder.Services.AddDbContext<ApplicationDbContext>(options =>
options.UseNpgsql(
configuration.GetConnectionString("Postgres")
)
);

builder.Services.AddHangfireWithPostgres(configuration);
builder.Services.AddMinio(configuration);
builder.Services.AddClickHouseAnalytics(configuration);

builder.Services.AddMediatR(cfg =>
cfg.RegisterServicesFromAssemblies(typeof(Program).Assembly));
builder.Services.AddFluentValidation();
builder.Services.AddJwtAuth(configuration);
builder.Services.AddAuthModule();
builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerWithAuth();

builder.Services.AddInfrastructure();
builder.Services.AddApplication();

builder.Services.AddTransient<ExceptionHandlerMiddleware>();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var db =
scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    await db.Database.MigrateAsync();
}

```

```

await app.Services.InitializeMinioAsync();

app.UseCors(cors =>
{
    cors.AllowAnyHeader();
    cors.AllowAnyMethod();
    cors.AllowAnyOrigin();
});

app.UseSwagger();
app.UseSwaggerUI();
app.MapHangfireDashboardWithAuth();

RecurringJob.AddOrUpdate<TrainingSessionStatusDispatcher>(
    "training-session-status-dispatcher",
    dispatcher => dispatcher.DispatchAsync(),
    "*/5 * * * *");

app.UseAuthorization();
app.UseMiddleware<ExceptionHandlingMiddleware>();

app.MapControllers();

app.Run();

```

Листинг Б.2 – Файл ExceptionHandlingMiddleware.cs

```

using System.Text.Json;
using SportManagementSystem.BuildingBlocks;
using SportManagementSystem.BuildingBlocks.Exceptions;
using SportManagementSystem.BuildingBlocks.Exceptions.Shared;

namespace SportManagementSystem.Middleware;

/// <summary>
/// Конвейер для обработки исключений.
/// </summary>
/// <remarks>
/// Этот конвейер перехватывает исключения, возникающие во время
/// обработки HTTP-запросов,
/// логирует их и возвращает соответствующий ответ клиенту.
/// </remarks>
public class
ExceptionHandlingMiddleware(ILogger<ExceptionHandlingMiddleware>
logger) : IMiddleware
{
    /// <summary>
    /// Обработывает входящий HTTP-запрос и перехватывает
    исключения.
    /// </summary>
    /// <param name="context">Контекст HTTP-запроса.</param>
    /// <param name="next">Делегат для передачи управления
    следующему компоненту в конвейере.</param>

```

```

    public async Task InvokeAsync(HttpContext context,
RequestDelegate next)
    {
        try
        {
            await next(context);
        }
        catch (Exception e)
        {
            logger.LogError(e, "Unhandled exception occurred:
{Message}", e.Message);
            await HandleExceptionAsync(context, e);
        }
    }

    /// <summary>
    /// Обработывает исключение и формирует ответ клиенту.
    /// </summary>
    /// <param name="httpContext">Контекст HTTP-запроса.</param>
    /// <param name="exception">Исключение, которое нужно
обработать.</param>
    private static async Task HandleExceptionAsync(HttpContext
httpContext, Exception exception)
    {
        var (statusCode, mbError) = GetMbError(exception);

        var response = MbResult<object>.Failure(mbError);

        httpContext.Response.ContentType = "application/json";
        httpContext.Response.StatusCode = statusCode;

        await
httpContext.Response.WriteAsync(JsonSerializer.Serialize(new
    {
        response.IsSuccess,
        response.Error!.Title,
        response.Error!.Status,
        response.Error!.Detail,
        response.Error!.Errors
    }));
    }

    private static (int StatusCode, MbError MbError)
GetMbError(Exception exception)
    {
        return exception switch
        {
            ValidationAppException validationException => (
                StatusCodes.Status422UnprocessableEntity,
                new MbError(
                    title: "Validation Error",
                    status:

```

```

        StatusCodes.Status422UnprocessableEntity,
            detail: validationException.Message,
            errors: validationException.Errors
        )
    ),
    BadRequestException badRequest => (
        StatusCodes.Status400BadRequest,
        new MbError(
            title: "Bad Request",
            status: StatusCodes.Status400BadRequest,
            detail: badRequest.Message
        )
    ),
    NotFoundException notFound => (
        StatusCodes.Status404NotFound,
        new MbError(
            title: "Not Found",
            status: StatusCodes.Status404NotFound,
            detail: notFound.Message
        )
    ),
    _ => (
        StatusCodes.Status500InternalServerError,
        new MbError(
            title: "Server Error",
            status:
StatusCodes.Status500InternalServerError,
            detail: exception.Message
        )
    )
};
}
}

```

Листинг Б.3 – Файл ApplicationHostExtensions.cs

```

using Hangfire;
using SportManagementSystem.BuildingBlocks.Hangfire;
using
SportManagementSystem.Modules.Scheduling.Application.Dispatchers;
using SportManagementSystem.Modules.Users.Domain.Services;
using SportManagementSystem.Modules.Users.Infrastructure.Services;

namespace SportManagementSystem.Extensions;

/// <summary>
/// Расширения для настройки сервисов приложения.
/// </summary>
/// <remarks>
/// Этот класс содержит методы для добавления сервисов приложения
/// в контейнер зависимостей.
/// </remarks>
public static class ApplicationHostExtensions

```

```

{
    /// <summary>
    /// Добавляет сервисы приложения в коллекцию сервисов.
    /// </summary>
    /// <param name="services">Коллекция сервисов.</param>
    public static void AddApplication(this IServiceCollection
services)
    {
        services.AddScoped<IJwtTokenService, JwtTokenService>();
        services.AddScoped<TrainingSessionStatusDispatcher>();
    }

    public static IEndpointRouteBuilder
MapHangfireDashboardWithAuth(this IEndpointRouteBuilder app)
    {
        app.MapHangfireDashboard("/hangfire", new DashboardOptions
        {
            Authorization = new[] { new
AllowAllDashboardAuthorizationFilter() }
        });

        return app;
    }
}

```

Листинг Б.4 – Файл AuthenticationExtensions.cs

```

using System.Text;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using SportManagementSystem.BuildingBlocks.Authentication.Hash;
using
SportManagementSystem.BuildingBlocks.Authentication.Hash.Interface
s;
using SportManagementSystem.BuildingBlocks.Authentication.Jwt;
using
SportManagementSystem.BuildingBlocks.Authentication.Jwt.Interfaces
;

namespace SportManagementSystem.Extensions;

/// <summary>
/// Расширения для настройки аутентификации в приложении.
/// </summary>
/// <remarks>
/// Этот класс содержит методы для добавления аутентификации JWT и
регистрации компонентов модуля аутентификации в контейнере
зависимостей.
/// </remarks>
public static class AuthenticationExtensions
{
    /// <summary>
    /// Добавляет аутентификацию JWT в коллекцию сервисов.

```

```

    /// </summary>
    /// <param name="services">Коллекция сервисов.</param>
    /// <param name="configuration">Конфигурация
приложения.</param>
    public static void AddJwtAuth(this IServiceCollection
services, IConfiguration configuration)
    {
        services.AddAuthorization();

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            var authSettings =
configuration.GetSection("Jwt");
            options.ClaimsIssuer = authSettings["Issuer"];
            options.SaveToken = true;
            options.TokenValidationParameters = new
TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidIssuer = authSettings["Issuer"],
                ValidateAudience = true,
                ValidAudience = authSettings["Audience"],
                ValidateLifetime = true,
                IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(authSettings["Key"])),
                ValidateIssuerSigningKey = true
            };
        });
    }

    /// <summary>
    /// Добавляет модуль аутентификации в коллекцию сервисов.
    /// </summary>
    /// <param name="services">Коллекция сервисов.</param>
    public static void AddAuthModule(this IServiceCollection
services)
    {
        services.AddSingleton<IJwtSettings, JwtSettings>();
        services.AddSingleton<IPasswordHasher,
BCryptPasswordHasher>();
    }
}

```

Листинг Б.5 – Файл ClickHouseAnalyticsExtensions.cs

```

using System.Net.Http.Headers;
using System.Text;
using Microsoft.Extensions.Options;
using
SportManagementSystem.Modules.Analytics.Application.EventHandlers;
using
SportManagementSystem.Modules.Analytics.Application.Repositories;

```

```

using
SportManagementSystem.Modules.Analytics.Infrastructure.ClickHouse;

namespace SportManagementSystem.Extensions;
public static class ClickHouseAnalyticsExtensions
{
    public static IServiceCollection AddClickHouseAnalytics(this
IServiceCollection services, IConfiguration configuration)
    {
        services.Configure<ClickHouseOptions>(configuration.GetSection("Cl
ickHouse"));

        services.AddHttpClient<ClickHouseAnalyticsClient>((provider,
client) =>
        {
            var options =
provider.GetRequiredService<IOptions<ClickHouseOptions>>().Value;
            client.BaseAddress = new
Uri(options.HttpUrl.TrimEnd('/') + "/");

            if (string.IsNullOrEmpty(options.Username))
            {
                return;
            }

            var authBytes =
Encoding.UTF8.GetBytes($"{options.Username}:{options.Password}");
            client.DefaultRequestHeaders.Authorization =
                new AuthenticationHeaderValue("Basic",
Convert.ToBase64String(authBytes));
        });

        services.AddScoped<IClickHouseAnalyticsClient>(provider =>
provider.GetRequiredService<ClickHouseAnalyticsClient>());
        services.AddScoped<IClientAnalyticsRepository>(provider =>
provider.GetRequiredService<ClickHouseAnalyticsClient>());
        services.AddScoped<IManagerAnalyticsRepository>(provider
=>
provider.GetRequiredService<ClickHouseAnalyticsClient>());
        services.AddScoped<IAalyticsEventWriter>(provider =>
provider.GetRequiredService<ClickHouseAnalyticsClient>());

        services.AddHostedService<ClickHouseAnalyticsInitializer>();

        return services;
    }
}

```

Приложение В
(обязательное)
Листинг клиентской части программы

Листинг В.1 – Файл webpack.config.js

```

const path = require('path');
const webpack = require('webpack');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const transpileNodeModules = [
  /@s77rt[\\/]react-native-date-picker/,
  /react-native-swiper/,
];

const webApiTarget = process.env.WEB_API_TARGET ||
'http://localhost:80';

module.exports = {
  entry: path.resolve(__dirname, 'index.web.js'),
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/',
    clean: true,
  },
  resolve: {
    extensions: ['.web.ts', '.web.tsx', '.web.js', '.ts', '.tsx',
'.js', '.json'],
    alias: {
      'react-native$': 'react-native-web',
    },
  },
  module: {
    rules: [
      {
        test: /\.ttf$/,
        type: 'asset/resource',
      },
      {
        test: /\. (png|jpe?g|gif|svg|webp) $/i,
        type: 'asset/resource',
      },
      {
        test: /\. [jt] sx? $/,
        exclude: (modulePath) => {
          if (!/node_modules/.test(modulePath)) {
            return false;
          }

          return !transpileNodeModules.some((moduleRegex) =>
moduleRegex.test(modulePath));
        },
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['module:@react-native/babel-preset'],
          },
        },
      },
    ],
  },
}
```

```

    },
  },
],
},
plugins: [
  new webpack.DefinePlugin({
    __DEV__: JSON.stringify(true),
  }),
  new HtmlWebpackPlugin({
    template: path.resolve(__dirname, 'web/index.html'),
  }),
],
devServer: {
  host: '0.0.0.0',
  port: 3000,
  hot: true,
  historyApiFallback: true,
  proxy: [
    {
      context: ['/v1'],
      target: webApiTarget,
      changeOrigin: true,
    },
  ],
},
};

```

Листинг В.2 – Файл NotFoundScreen.tsx

```

import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { AppButton } from '../../shared/ui/AppButton';
import { AppLogo } from '../../shared/ui/AppLogo';
import { ScreenContainer } from '../../shared/ui/ScreenContainer';

type NotFoundScreenProps = {
  onGoHome: () => void;
};

export function NotFoundScreen({ onGoHome }: NotFoundScreenProps) {
  return (
    <ScreenContainer>
      <View style={styles.card}>
        <Text style={styles.title}>Страница не найдена</Text>
        <Text style={styles.text}>Проверьте адрес страницы или
вернитесь на главную.</Text>
        <AppButton title="На главную" icon={<AppLogo compact
inverted markSize={22} />} onPress={onGoHome} />
      </View>
    </ScreenContainer>
  );
}

```

```

const styles = StyleSheet.create({
  card: {
    backgroundColor: 'ffffff',
    borderRadius: 14,
    padding: 18,
    gap: 10,
    borderWidth: 1,
    borderColor: 'd6dfef',
    maxWidth: 620,
    alignSelf: 'center',
    width: '100%',
  },
  title: {
    fontSize: 24,
    lineHeight: 30,
    fontWeight: '700',
    color: '#15223a',
  },
  text: {
    color: '#334763',
    fontSize: 15,
    lineHeight: 22,
  },
});

```

Листинг В.3 – Файл AdminBranchesScreen.tsx

```

import React from 'react';
import { Pressable, StyleSheet, Text, View } from 'react-native';
import { Session } from '../../entities/session/model/session';
import { AdminBranchManagementScreen } from
'../../features/branches/ui/AdminBranchManagementScreen';
import { AdminBranchesListScreen } from
'../../features/branches/ui/AdminBranchesListScreen';
import { AdminCreateBranchScreen } from
'../../features/branches/ui/AdminCreateBranchScreen';
import { AdminEquipmentManagementScreen } from
'../../features/branches/ui/AdminEquipmentManagementScreen';
import { AdminRoomManagementScreen } from
'../../features/branches/ui/AdminRoomManagementScreen';
import { ScreenContainer } from '../../shared/ui/ScreenContainer';

type AdminBranchesScreenProps = {
  session: Session;
  routePath: string;
  onNavigate: (nextPath: `/admin-branches${string}`) => void;
  onShowToast: (message: string) => void;
  canCreateBranch: boolean;
};

type AdminRoute =
  | { mode: 'list' }

```

```

    | { mode: 'create' }
    | { mode: 'manage'; branchId: number }
    | { mode: 'room'; branchId: number; roomId: number }
    | { mode: 'equipment'; branchId: number; roomId: number;
equipmentId: number };

export function AdminBranchesScreen({
  session,
  routePath,
  onNavigate,
  onShowToast,
  canCreateBranch,
}: AdminBranchesScreenProps) {
  const route = parseAdminRoute(routePath);
  const effectiveRoute = route.mode === 'create' &&
!canCreateBranch ? { mode: 'list' as const } : route;

  if (effectiveRoute.mode === 'manage') {
    return (
      <AdminBranchManagementScreen
        token={session.token}
        branchId={effectiveRoute.branchId}
        onBack={() => onNavigate('/admin-branches')}
        onOpenRoom={roomId => onNavigate(`/admin-
branches/${effectiveRoute.branchId}/rooms/${roomId}`)}
        onShowToast={onShowToast}
      />
    );
  }

  if (effectiveRoute.mode === 'room') {
    return (
      <AdminRoomManagementScreen
        token={session.token}
        branchId={effectiveRoute.branchId}
        roomId={effectiveRoute.roomId}
        onBack={() => onNavigate(`/admin-
branches/${effectiveRoute.branchId}`)}
        onOpenEquipment={equipmentId =>
          onNavigate(`/admin-
branches/${effectiveRoute.branchId}/rooms/${effectiveRoute.roomId}
/equipments/${equipmentId}`)}
        onShowToast={onShowToast}
      />
    );
  }

  if (effectiveRoute.mode === 'equipment') {
    return (
      <AdminEquipmentManagementScreen
        token={session.token}

```

```

        branchId={effectiveRoute.branchId}
        roomId={effectiveRoute.roomId}
        equipmentId={effectiveRoute.equipmentId}
        onBack={() => onNavigate(`/admin-
branches/${effectiveRoute.branchId}/rooms/${effectiveRoute.roomId}
`)}
        onShowToast={onShowToast}
      />
    );
  }

  return (
    <ScreenContainer>
      <View style={styles.panelHeader}>
        <Text style={styles.panelTitle}>Филиалы</Text>
      </View>

      <View style={styles.switch}>
        <Pressable
          style={[styles.switchButton, effectiveRoute.mode ===
'list' ? styles.switchActive : undefined]}
          onPress={() => onNavigate('/admin-branches')}>
          <Text style={styles.switchText, effectiveRoute.mode ===
'list' ? styles.switchTextActive : undefined}>
            Все филиалы
          </Text>
        </Pressable>
        {canCreateBranch ? (
          <Pressable
            style={[styles.switchButton, effectiveRoute.mode ===
'create' ? styles.switchActive : undefined]}
            onPress={() => onNavigate('/admin-branches/create')}>
            <Text style={styles.switchText, effectiveRoute.mode ===
'create' ? styles.switchTextActive : undefined}>
              Создать филиал
            </Text>
          </Pressable>
        ) : null}
      </View>

      {effectiveRoute.mode === 'create' ? (
        <AdminCreateBranchScreen
          token={session.token}
          onCreate={() => onNavigate('/admin-branches')}
          onShowToast={onShowToast}
        />
      ) : (
        <AdminBranchesListScreen
          token={session.token}
          reloadKey={0}
          onOpenBranch={branchId => onNavigate(`/admin-
branches/${branchId}`)}

```

```

        />
    })
</ScreenContainer>
);
}

function parseAdminRoute(pathname: string): AdminRoute {
    if (pathname === '/admin-branches/create') {
        return { mode: 'create' };
    }

    const match = pathname.match(/^\/admin-branches\/(\d+)\$/);
    if (match) {
        return { mode: 'manage', branchId: Number(match[1]) };
    }

    const roomMatch = pathname.match(/^\/admin-branches\/(\d+)\rooms\/(\d+)\$/);
    if (roomMatch) {
        return { mode: 'room', branchId: Number(roomMatch[1]), roomId: Number(roomMatch[2]) };
    }

    const equipmentMatch = pathname.match(/^\/admin-branches\/(\d+)\rooms\/(\d+)\equipments\/(\d+)\$/);
    if (equipmentMatch) {
        return {
            mode: 'equipment',
            branchId: Number(equipmentMatch[1]),
            roomId: Number(equipmentMatch[2]),
            equipmentId: Number(equipmentMatch[3]),
        };
    }

    return { mode: 'list' };
}

const styles = StyleSheet.create({
    panelHeader: {
        marginBottom: 10,
    },
    panelTitle: {
        fontSize: 24,
        lineHeight: 30,
        fontWeight: '700',
        color: '#15223a',
    },
    switch: {
        flexDirection: 'row',
        backgroundColor: '#e7edf8',
        borderRadius: 12,
        padding: 4,
    },
});

```

```
        marginBottom: 12,  
        maxWidth: 620,  
        width: '100%',  
        alignSelf: 'center',  
    },  
    switchButton: {  
        flex: 1,  
        borderRadius: 10,  
        paddingVertical: 10,  
        alignItems: 'center',  
    },  
    switchActive: {  
        backgroundColor: '#0d4aa6',  
    },  
    switchText: {  
        color: '#0d4aa6',  
        fontSize: 15,  
        fontWeight: '600',  
    },  
    switchTextActive: {  
        color: '#ffffff',  
    },  
    },  
    ));
```

Приложение Г
(рекомендуемое)
Листинг модульных тестов

Листинг Г.1 – Файл AssetsValidatorsTests.cs

```

using
SportManagementSystem.Modules.Assets.Application.Commands.CreateBranch;
using
SportManagementSystem.Modules.Assets.Application.Commands.CreateEquipment;
using
SportManagementSystem.Modules.Assets.Application.Commands.CreateRoom;
using SportManagementSystem.Modules.Assets.Contracts.Request;
using SportManagementSystem.Modules.Assets.Domain.Enums;

namespace SportManagementSystem.Tests.Modules.Assets.Unit;

public class AssetsValidatorsTests
{
    [Fact]
    public void
CreateBranchValidator_WhenRequestInvalid_ReturnsErrors()
    {
        var validator = new CreateBranchValidator();
        var message = new CreateBranchMessage(new
CreateBranchRequest
        {
            Name = "",
            Address = ""
        });

        var result = validator.Validate(message);

        Assert.False(result.IsValid);
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Name");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Address");
    }

    [Fact]
    public void
CreateRoomValidator_WhenRequestInvalid_ReturnsErrors()
    {
        var validator = new CreateRoomValidator();
        var message = new CreateRoomMessage(new CreateRoomRequest
        {
            BranchId = 0,
            Name = "A",
            Capacity = 0
        });

        var result = validator.Validate(message);
    }
}

```

```

        Assert.False(result.IsValid);
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.BranchId");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Name");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Capacity");
    }

    [Fact]
    public void
CreateEquipmentValidator_WhenRequestInvalid_ReturnsErrors()
    {
        var validator = new CreateEquipmentValidator();
        var message = new CreateEquipmentMessage(new
CreateEquipmentRequest
        {
            RoomId = 0,
            Name = "",
            Quantity = 0,
            Condition = (EEquipmentCondition)99
        });

        var result = validator.Validate(message);

        Assert.False(result.IsValid);
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.RoomId");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Name");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Quantity");
        Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Condition");
    }

    [Fact]
    public void
CreateEquipmentValidator_WhenRequestValid_HasNoErrors()
    {
        var validator = new CreateEquipmentValidator();
        var message = new CreateEquipmentMessage(new
CreateEquipmentRequest
        {
            RoomId = 1,
            Name = "Bike",
            Quantity = 2,
            Condition = EEquipmentCondition.New
        });

        var result = validator.Validate(message);

```

```

        Assert.True(result.IsValid);
    }
}

```

Листинг Г.2 – Файл CreateBranchHandlerTests.cs

```

using Moq;
using
SportManagementSystem.Modules.Assets.Application.Commands.CreateBranch;
using SportManagementSystem.Modules.Assets.Contracts.Request;
using SportManagementSystem.Modules.Assets.Domain.Entities;
using SportManagementSystem.Modules.Assets.Domain.Repositories;

namespace SportManagementSystem.Tests.Modules.Assets.Unit;

public class CreateBranchHandlerTests
{
    private readonly Mock<IBranchRepository> _branchRepository;
    private readonly CreateBranchHandler _handler;

    public CreateBranchHandlerTests()
    {
        _branchRepository = new Mock<IBranchRepository>();
        _handler = new
CreateBranchHandler(_branchRepository.Object);
    }

    [Fact]
    public async Task Handle_WhenRequestValid_CreatesBranch()
    {
        DbBranch? created = null;
        _branchRepository
            .Setup(x => x.CreateAsync(It.IsAny<DbBranch>(),
It.IsAny<CancellationToken>()))
            .Callback<DbBranch, CancellationToken>((entity, _) =>
created = entity)
            .ReturnsAsync((DbBranch entity, CancellationToken _)
=>
        {
            entity.Id = 12;
            return entity;
        });

        var result = await _handler.Handle(new
CreateBranchMessage(new CreateBranchRequest
        {
            Name = "Central",
            Address = "Main street"
        }), CancellationToken.None);

        Assert.True(result.IsSuccess);
        Assert.Equal(12, result.Data);
    }
}

```

```

        Assert.NotNull(created);
        Assert.Equal("Central", created!.Name);
    }
}

```

Листинг Г.3 – Файл CreateEquipmentHandlerTests.cs

```

using Microsoft.AspNetCore.Http;
using Moq;
using
SportManagementSystem.Modules.Assets.Application.Commands.CreateEq
uipment;
using SportManagementSystem.Modules.Assets.Contracts.Request;
using SportManagementSystem.Modules.Assets.Domain.Entities;
using SportManagementSystem.Modules.Assets.Domain.Enums;
using SportManagementSystem.Modules.Assets.Domain.Repositories;

namespace SportManagementSystem.Tests.Modules.Assets.Unit;

public class CreateEquipmentHandlerTests
{
    private readonly Mock<IEquipmentRepository>
    _equipmentRepository;
    private readonly Mock<IRoomRepository> _roomRepository;
    private readonly CreateEquipmentHandler _handler;

    public CreateEquipmentHandlerTests()
    {
        _equipmentRepository = new Mock<IEquipmentRepository>();
        _roomRepository = new Mock<IRoomRepository>();
        _handler = new
CreateEquipmentHandler(_equipmentRepository.Object,
    _roomRepository.Object);
    }

    [Fact]
    public async Task Handle_WhenRoomMissing_ReturnsConflict()
    {
        _roomRepository.Setup(x => x.ExistsAsync(It.IsAny<long>()),
It.IsAny<CancellationToken>())).ReturnsAsync(false);

        var result = await _handler.Handle(new
CreateEquipmentMessage(new CreateEquipmentRequest
        {
            RoomId = 8,
            Name = "Treadmill",
            Quantity = 2,
            Condition = EEquipmentCondition.New
        })), CancellationToken.None);

        Assert.False(result.IsSuccess);
        Assert.NotNull(result.Error);
        Assert.Equal(StatusCodes.Status409Conflict,

```

```

result.Error!.Status);
    _equipmentRepository.Verify(x =>
x.CreateAsync(It.IsAny<DbEquipment>()),
It.IsAny<CancellationToken>()), Times.Never);
    }

    [Fact]
    public async Task Handle_WhenRequestValid_CreatesEquipment()
    {
        DbEquipment? created = null;
        _roomRepository.Setup(x => x.ExistsAsync(It.IsAny<long>()),
It.IsAny<CancellationToken>())) .ReturnsAsync(true);
        _equipmentRepository
            .Setup(x => x.CreateAsync(It.IsAny<DbEquipment>()),
It.IsAny<CancellationToken>()))
            .Callback<DbEquipment, CancellationToken>((entity, _)
=> created = entity)
            .ReturnsAsync((DbEquipment entity, CancellationToken
_) =>
            {
                entity.Id = 77;
                return entity;
            });

        var result = await _handler.Handle(new
CreateEquipmentMessage(new CreateEquipmentRequest
        {
            RoomId = 3,
            Name = "Bike",
            Quantity = 5,
            Condition = EEquipmentCondition.Good
        }), CancellationToken.None);

        Assert.True(result.IsSuccess);
        Assert.Equal(77, result.Data);
        Assert.NotNull(created);
        Assert.Equal("Bike", created!.Name);
        Assert.Equal(EEquipmentCondition.Good, created.Condition);
    }
}

```

Листинг Г.4 – Файл CreateRoomHandlerTests.cs

```

using Microsoft.AspNetCore.Http;
using Moq;
using
SportManagementSystem.Modules.Assets.Application.Commands.CreateRo
om;
using SportManagementSystem.Modules.Assets.Contracts.Request;
using SportManagementSystem.Modules.Assets.Domain.Entities;
using SportManagementSystem.Modules.Assets.Domain.Enums;
using SportManagementSystem.Modules.Assets.Domain.Repositories;

```

```

namespace SportManagementSystem.Tests.Modules.Assets.Unit;

public class CreateRoomHandlerTests
{
    private readonly Mock<IRoomRepository> _roomRepository;
    private readonly Mock<IBranchRepository> _branchRepository;
    private readonly CreateRoomHandler _handler;

    public CreateRoomHandlerTests()
    {
        _roomRepository = new Mock<IRoomRepository>();
        _branchRepository = new Mock<IBranchRepository>();
        _handler = new CreateRoomHandler(_roomRepository.Object,
        _branchRepository.Object);
    }

    [Fact]
    public async Task Handle_WhenBranchMissing_ReturnsConflict()
    {
        _branchRepository.Setup(x =>
x.ExistsAsync(It.IsAny<long>()),
It.IsAny<CancellationToken>())) .ReturnsAsync(false);

        var result = await _handler.Handle(new
CreateRoomMessage(new CreateRoomRequest
        {
            BranchId = 99,
            Name = "Room A",
            Capacity = 10
        })), CancellationToken.None));

        Assert.False(result.IsSuccess);
        Assert.NotNull(result.Error);
        Assert.Equal(StatusCodes.Status409Conflict,
result.Error!.Status);
        _roomRepository.Verify(x =>
x.CreateAsync(It.IsAny<DbRoom>(), It.IsAny<CancellationToken>()),
Times.Never);
    }

    [Fact]
    public async Task Handle_WhenRequestValid_CreatesRoom()
    {
        DbRoom? created = null;
        _branchRepository.Setup(x =>
x.ExistsAsync(It.IsAny<long>()),
It.IsAny<CancellationToken>())) .ReturnsAsync(true);
        _roomRepository
            .Setup(x => x.CreateAsync(It.IsAny<DbRoom>(),
It.IsAny<CancellationToken>()))
            .Callback<DbRoom, CancellationToken>((entity, _) =>
created = entity)

```

```

        .ReturnsAsync((DbRoom entity, CancellationToken _) =>
        {
            entity.Id = 33;
            return entity;
        });

        var result = await _handler.Handle(new
        CreateRoomMessage(new CreateRoomRequest
        {
            BranchId = 5,
            Name = "Room B",
            Capacity = 20
        }), CancellationToken.None);

        Assert.True(result.IsSuccess);
        Assert.Equal(33, result.Data);
        Assert.NotNull(created);
        Assert.Equal(ERoomStatus.Available, created!.Status);
    }
}

```

Листинг Г.5 – Файл ClientsValidatorsTests.cs

```

using
SportManagementSystem.Modules.Clients.Application.Commands.CreateB
ooking;
using
SportManagementSystem.Modules.Clients.Application.Commands.CreateM
embership;
using SportManagementSystem.Modules.Clients.Contracts.Requests;

namespace SportManagementSystem.Tests.Modules.Clients.Unit;

public class ClientsValidatorsTests
{
    [Fact]
    public void
    CreateBookingValidator_WhenRequestInvalid_ReturnsErrors()
    {
        var validator = new CreateBookingValidator();
        var message = new CreateBookingMessage(0, new
        CreateBookingRequest
        {
            SessionId = 0,
            Booked = DateTime.UtcNow.AddMinutes(-5)
        });

        var result = validator.Validate(message);

        Assert.False(result.IsValid);
        Assert.Contains(result.Errors, e => e.PropertyName ==
        "ClientId");
        Assert.Contains(result.Errors, e => e.PropertyName ==

```

```

"Request.SessionId");
    Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.Booking");
}

[Fact]
public void
CreateBookingValidator_WhenRequestValid_HasNoErrors()
{
    var validator = new CreateBookingValidator();
    var message = new CreateBookingMessage(1, new
CreateBookingRequest
    {
        SessionId = 5,
        Booked = DateTime.UtcNow.AddMinutes(15)
    });

    var result = validator.Validate(message);

    Assert.True(result.IsValid);
}

[Fact]
public void
CreateMembershipValidator_WhenRequestInvalid_ReturnsErrors()
{
    var validator = new CreateMembershipValidator();
    var today = DateOnly.FromDateTime(DateTime.UtcNow);
    var message = new CreateMembershipMessage(1, new
CreateMembershipRequest
    {
        SportServiceId = 0,
        StartDate = today,
        EndDate = today.AddDays(1),
        RemainingVisits = 0
    });

    var result = validator.Validate(message);

    Assert.False(result.IsValid);
    Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.SportServiceId");
    Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.StartDate");
    Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.EndDate");
    Assert.Contains(result.Errors, e => e.PropertyName ==
"Request.RemainingVisits");
}

[Fact]
public void

```

```
CreateMembershipValidator_WhenRequestValid_HasNoErrors()
{
    var validator = new CreateMembershipValidator();
    var today = DateOnly.FromDateTime(DateTime.UtcNow);
    var message = new CreateMembershipMessage(1, new
CreateMembershipRequest
    {
        SportServiceId = 5,
        StartDate = today.AddDays(1),
        EndDate = today.AddDays(2),
        RemainingVisits = 10
    });

    var result = validator.Validate(message);

    Assert.True(result.IsValid);
}
}
```

Приложение Д
(рекомендуемое)
Листинг интеграционных тестов

Листинг Д.1 – Файл PostgreSQLCollection.cs

```
namespace SportManagementSystem.Tests.Infrastructure;

[CollectionDefinition(Name)]
public sealed class PostgreSQLCollection :
ICollectionFixture<PostgreSQLTestContainerFixture>
{
    public const string Name = "PostgreSQL";
}
```

Листинг Д.2 – Файл PostgreSQLTestContainerFixture.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using SportManagementSystem.Data;
using Testcontainers.PostgreSql;
namespace SportManagementSystem.Tests.Infrastructure;

public sealed class PostgreSQLTestContainerFixture :
IAsyncLifetime
{
    private readonly PostgreSqlContainer _container = new
PostgreSqlBuilder()
        .WithImage("postgres:18.1")
        .WithDatabase("sport_management_tests")
        .WithUsername("postgres")
        .WithPassword("password")
        .Build();

    public ApplicationDbContext CreateDbContext()
    {
        var options = new
DbContextOptionsBuilder<ApplicationDbContext>()
            .UseNpgsql(_container.GetConnectionString())
            .ConfigureWarnings(w =>
w.Ignore(RelationalEventId.PendingModelChangesWarning))
            .Options;

        return new ApplicationDbContext(options);
    }

    public async Task InitializeAsync()
    {
        await _container.StartAsync();

        await using var context = CreateDbContext();
        await context.Database.MigrateAsync();
    }

    public async Task DisposeAsync()
    {
        await _container.DisposeAsync();
    }
}
```

```

    }
}

```

Листинг Д.3 – Файл AssetsRepositoriesIntegrationTests.cs

```

using SportManagementSystem.Modules.Assets.Domain.Entities;
using SportManagementSystem.Modules.Assets.Domain.Enums;
using
SportManagementSystem.Modules.Assets.Infrastructure.Repositories;
using SportManagementSystem.Tests.Infrastructure;

namespace SportManagementSystem.Tests.Modules.Assets.Integration;

[Collection(PostgreSqlCollection.Name)]
public class
AssetsRepositoriesIntegrationTests(PostgreSqlTestContainerFixture
fixture)
{
    [Fact]
    public async Task BranchRepository_CreateAndGetById_Works()
    {
        await using var db = fixture.CreateDbContext();
        var repository = new BranchRepository(db);

        var created = await repository.CreateAsync(new DbBranch
        {
            Name = "North",
            Address = "Street 9"
        }, CancellationToken.None);

        var fetched = await repository.GetByIdAsync(created.Id,
CancellationToken.None);

        Assert.True(created.Id > 0);
        Assert.NotNull(fetched);
        Assert.Equal("North", fetched!.Name);
    }

    [Fact]
    public async Task RoomRepository_GetByBranch_ReturnsRooms()
    {
        await using var db = fixture.CreateDbContext();
        var branchRepository = new BranchRepository(db);
        var roomRepository = new RoomRepository(db);

        var branch = await branchRepository.CreateAsync(new
DbBranch
        {
            Name = "East",
            Address = "Street 5"
        }, CancellationToken.None);

        await roomRepository.CreateAsync(new DbRoom

```

```

        {
            BranchId = branch.Id,
            Name = "Room X",
            Capacity = 12,
            Status = ERoomStatus.Available
        }, CancellationToken.None);

        var rooms = await
roomRepository.GetByBranchAsync(branch.Id,
CancellationToken.None);

        Assert.NotNull(rooms);
        Assert.Single(rooms!);
        Assert.Equal(branch.Id, rooms![0].BranchId);
    }

    [Fact]
    public async Task EquipmentRepository_CreateAndGetById_Works()
    {
        await using var db = fixture.CreateDbContext();
        var branchRepository = new BranchRepository(db);
        var roomRepository = new RoomRepository(db);
        var equipmentRepository = new EquipmentRepository(db);

        var branch = await branchRepository.CreateAsync(new
DbBranch
        {
            Name = "South",
            Address = "Street 3"
        }, CancellationToken.None);

        var room = await roomRepository.CreateAsync(new DbRoom
        {
            BranchId = branch.Id,
            Name = "Room Z",
            Capacity = 8,
            Status = ERoomStatus.Available
        }, CancellationToken.None);

        var created = await equipmentRepository.CreateAsync(new
DbEquipment
        {
            RoomId = room.Id,
            Name = "Rope",
            Quantity = 3,
            Condition = EEquipmentCondition.New
        }, CancellationToken.None);

        var fetched = await
equipmentRepository.GetByIdAsync(created.Id,
CancellationToken.None);

```

```

        Assert.True(created.Id > 0);
        Assert.NotNull(fetched);
        Assert.Equal(room.Id, fetched!.RoomId);
        Assert.Equal("Rope", fetched.Name);
    }
}

```

Листинг Д.4 – Файл ClientsRepositoriesIntegrationTests.cs

```

using SportManagementSystem.Modules.Assets.Domain.Entities;
using SportManagementSystem.Modules.Clients.Domain.Entities;
using SportManagementSystem.Modules.Clients.Domain.Enums;
using
SportManagementSystem.Modules.Clients.Infrastructure.Repositories;
using SportManagementSystem.Modules.Scheduling.Domain.Entities;
using SportManagementSystem.Modules.Scheduling.Domain.Enums;
using SportManagementSystem.Modules.Services.Domain.Entities;
using SportManagementSystem.Modules.Services.Domain.Enums;
using SportManagementSystem.Modules.Users.Domain.Entities;
using SportManagementSystem.Modules.Users.Domain.Enums;
using SportManagementSystem.Tests.Infrastructure;

namespace SportManagementSystem.Tests.Modules.Clients.Integration;

[Collection(PostgreSqlCollection.Name)]
public class
ClientsRepositoriesIntegrationTests(PostgreSqlTestContainerFixture
fixture)
{
    [Fact]
    public async Task
MembershipRepository_CreateAndGetById_Works()
    {
        await using var db = fixture.CreateDbContext();
        var membershipRepository = new MembershipRepository(db);

        var client = db.Clients.Add(new DbClient
        {
            FirstName = "John",
            LastName = "Doe",
            BirthDate = new DateOnly(1995, 1, 1),
            Phone = "+12345678901",
            RegisterDate = DateTime.UtcNow
        }).Entity;

        var service = db.SportServices.Add(new DbSportService
        {
            Code = $"svc-{Guid.NewGuid():N}".Substring(0, 12),
            Name = "Yoga",
            Category = EServiceCategory.Fitness,
            IsActive = true,
            Created = DateTime.UtcNow
        }).Entity;
    }
}

```

```

        await db.SaveChangesAsync(CancellationToken.None);

        var created = await membershipRepository.CreateAsync(new
DbMembership
        {
            ClientId = client.Id,
            SportServiceId = service.Id,
            StartDate =
DateOnly.FromDateTime(DateTime.UtcNow.AddDays(1)),
            EndDate =
DateOnly.FromDateTime(DateTime.UtcNow.AddDays(31)),
            TotalVisits = 0,
            RemainingVisits = 8,
            Status = EMembershipStatus.Active
        }, CancellationToken.None);

        var fetched = await
membershipRepository.GetByIdAsync(created.Id,
CancellationToken.None);

        Assert.True(created.Id > 0);
        Assert.NotNull(fetched);
        Assert.Equal(client.Id, fetched!.ClientId);
        Assert.Equal(service.Id, fetched.SportServiceId);
        Assert.Equal(EMembershipStatus.Active, fetched.Status);
    }

    [Fact]
    public async Task BookingRepository_CreateAndGetById_Works()
    {
        await using var db = fixture.CreateDbContext();
        var bookingRepository = new BookingRepository(db);

        var branch = db.Branches.Add(new DbBranch
        {
            Name = "Main",
            Address = "Street 1"
        }).Entity;

        await db.SaveChangesAsync(CancellationToken.None);

        var staff = db.Staffs.Add(new DbStaff
        {
            FirstName = "Alice",
            LastName = "Trainer",
            Phone = "+12345678902",
            BirthDate = new DateOnly(1990, 2, 2),
            BranchId = branch.Id
        }).Entity;

        var service = db.SportServices.Add(new DbSportService

```

```

        {
            Code = $"svc-{Guid.NewGuid():N}".Substring(0, 12),
            Name = "Pilates",
            Category = EServiceCategory.Fitness,
            IsActive = true,
            Created = DateTime.UtcNow
        }).Entity;

        await db.SaveChangesAsync(CancellationToken.None);

        var session = db.TrainingSessions.Add(new
        DbTrainingSession
        {
            SportServiceId = service.Id,
            StartedDate = DateTime.UtcNow.AddHours(2),
            EndedDate = DateTime.UtcNow.AddHours(3),
            TrainerId = staff.Id,
            Status = ESessionStatus.Planned
        }).Entity;

        await db.SaveChangesAsync(CancellationToken.None);

        var created = await bookingRepository.CreateAsync(new
        DbBooking
        {
            ClientId = null,
            SessionId = session.Id,
            Booked = DateTime.UtcNow.AddHours(1),
            Status = EBookingStatus.Booked
        }, CancellationToken.None);

        var fetched = await
        bookingRepository.GetByIdAsync(created.Id,
        CancellationToken.None);

        Assert.True(created.Id > 0);
        Assert.NotNull(fetched);
        Assert.Equal(session.Id, fetched!.SessionId);
        Assert.Equal(EBookingStatus.Booked, fetched.Status);
    }
}

```

Листинг Д.5 – Файл ImagesRepositoryIntegrationTests.cs

```

using SportManagementSystem.Modules.Images.Domain.Entities;
using
SportManagementSystem.Modules.Images.Infrastructure.Repositories;
using SportManagementSystem.Tests.Infrastructure;

namespace SportManagementSystem.Tests.Modules.Images.Integration;

[Collection(PostgreSqlCollection.Name)]
public class

```

```

ImagesRepositoryIntegrationTests(PostgreSqlTestContainerFixture
fixture)
{
    [Fact]
    public async Task ImageRepository_CreateAndGetById_Works()
    {
        await using var db = fixture.CreateDbContext();
        var repository = new ImageRepository(db);

        var id = Guid.NewGuid();
        var created = await repository.CreateAsync(new DbImage
        {
            Id = id,
            ObjectName = "file.jpg",
            Bucket = "images",
            FileName = "file.jpg",
            ContentType = "image/jpeg",
            Length = 123
        }, CancellationToken.None);

        var fetched = await repository.GetByIdAsync(id,
CancellationToken.None);

        Assert.Equal(id, created.Id);
        Assert.NotNull(fetched);
        Assert.Equal("file.jpg", fetched!.FileName);
    }

    [Fact]
    public async Task ImageRepository_Delete_RemovesEntity()
    {
        await using var db = fixture.CreateDbContext();
        var repository = new ImageRepository(db);

        var id = Guid.NewGuid();
        await repository.CreateAsync(new DbImage
        {
            Id = id,
            ObjectName = "file2.jpg",
            Bucket = "images",
            FileName = "file2.jpg",
            ContentType = "image/jpeg",
            Length = 10
        }, CancellationToken.None);

        await repository.DeleteAsync(id, CancellationToken.None);
        var exists = await repository.ExistsAsync(id,
CancellationToken.None);

        Assert.False(exists);
    }
}

```

Приложение Е
(рекомендуемое)
Графическая часть

Рисунок Е.1 – Диаграмма вариантов использования

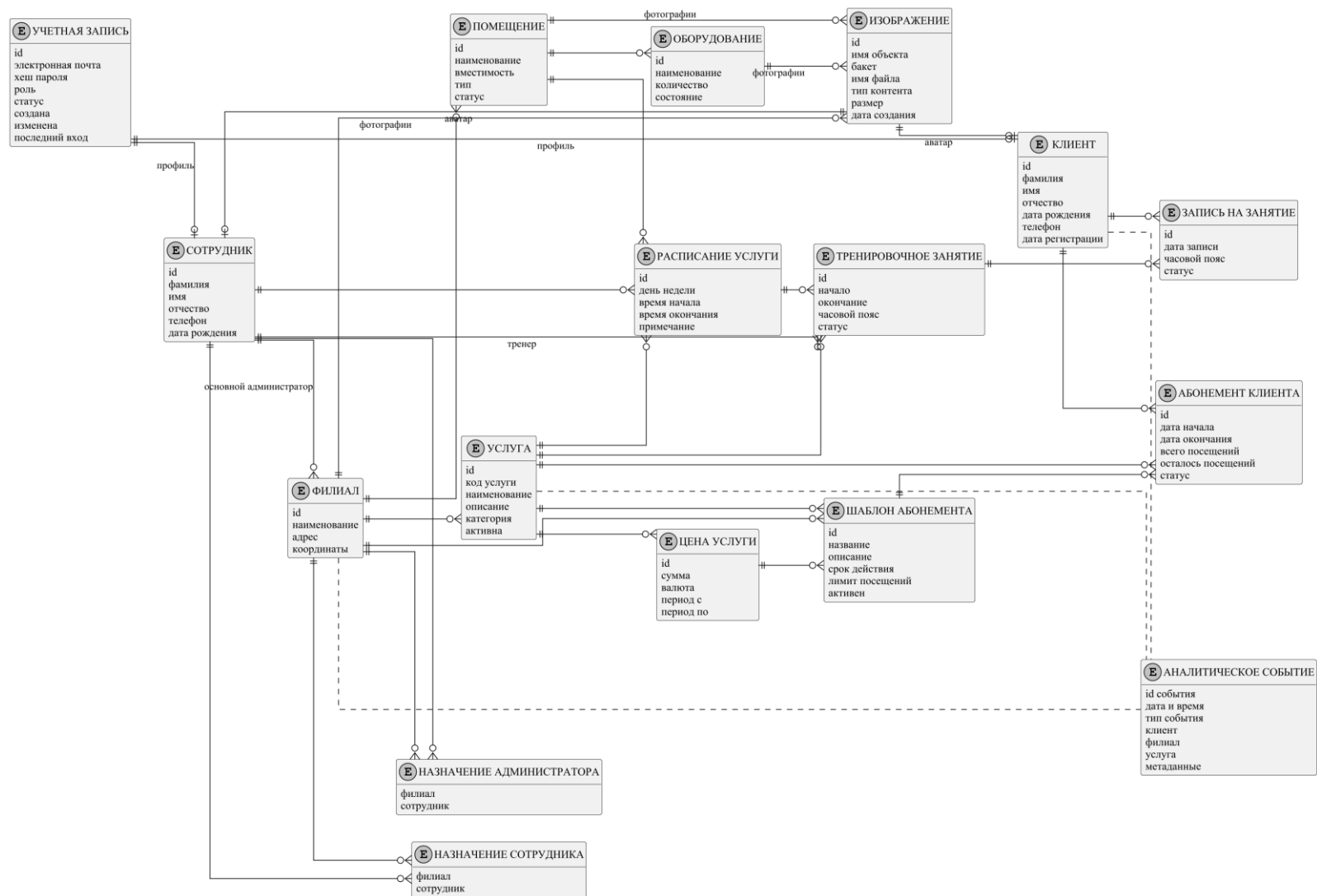


Рисунок Е.2 – Концептуальная модель базы данных

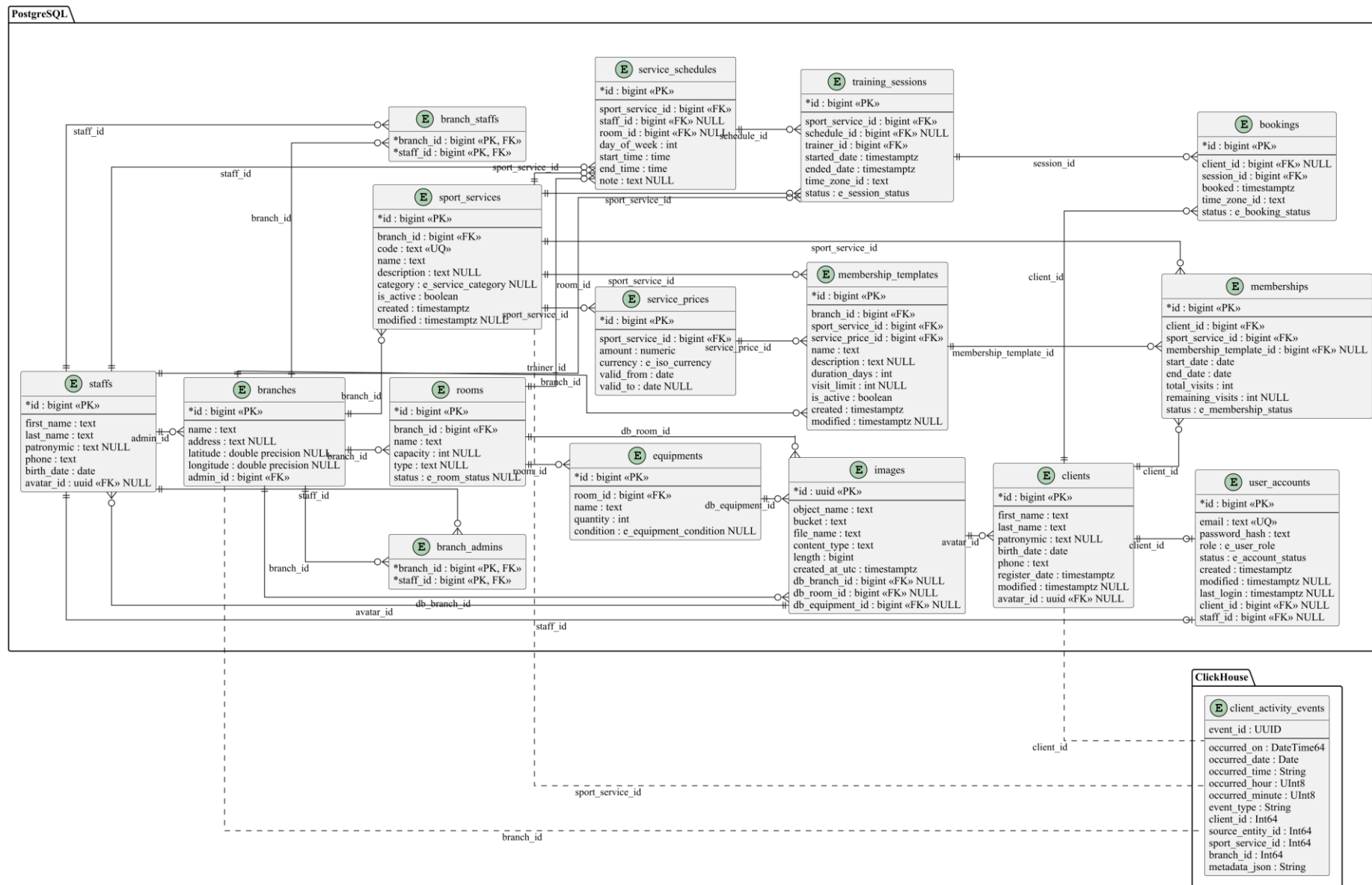
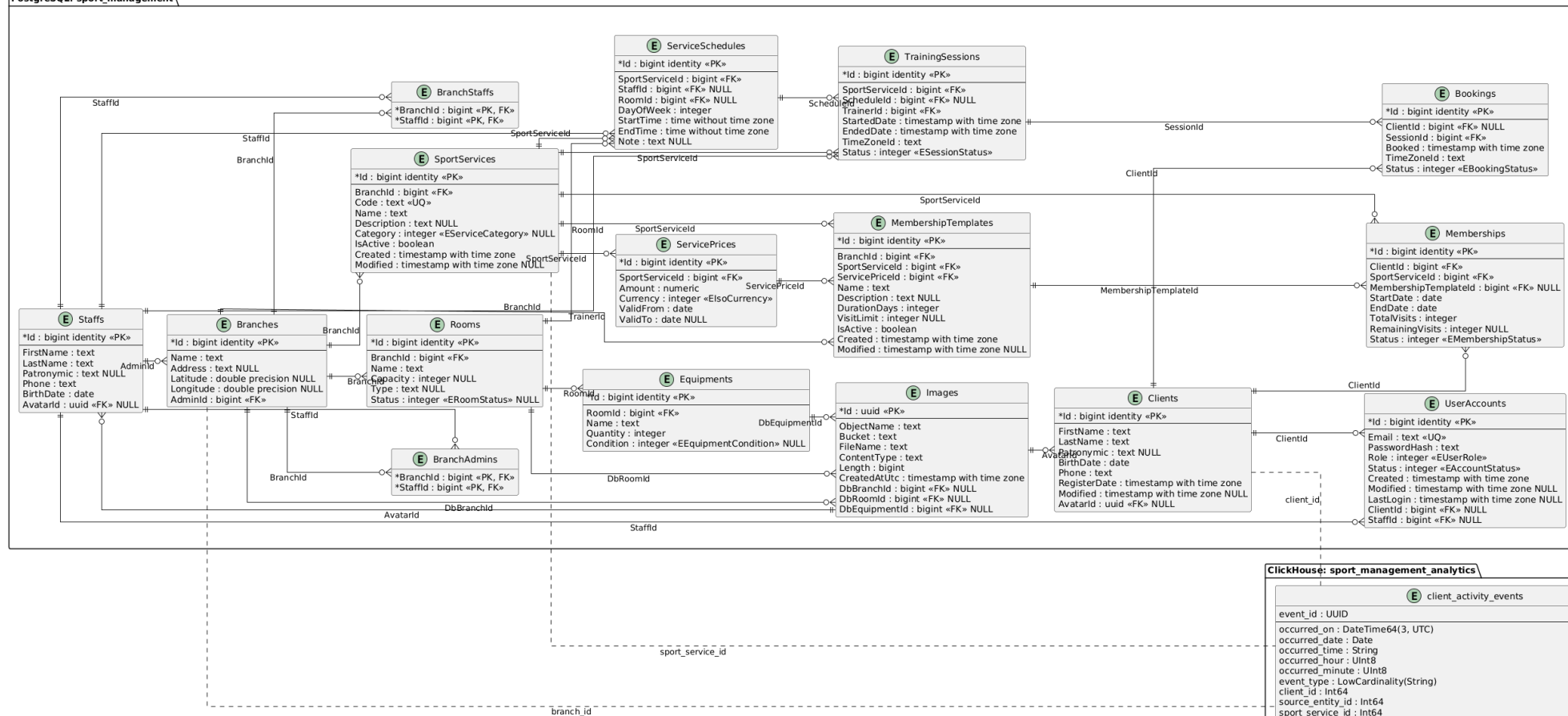


Рисунок Е.3 – Логическая модель базы данных

PostgreSQL: sport_management



ClickHouse: sport_management_analytics

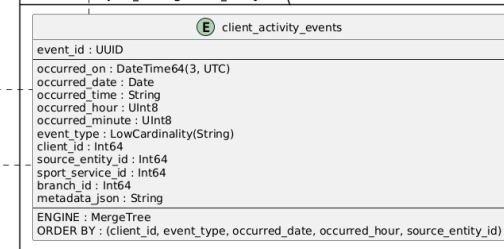


Рисунок Е.4 – Физическая модель базы данных

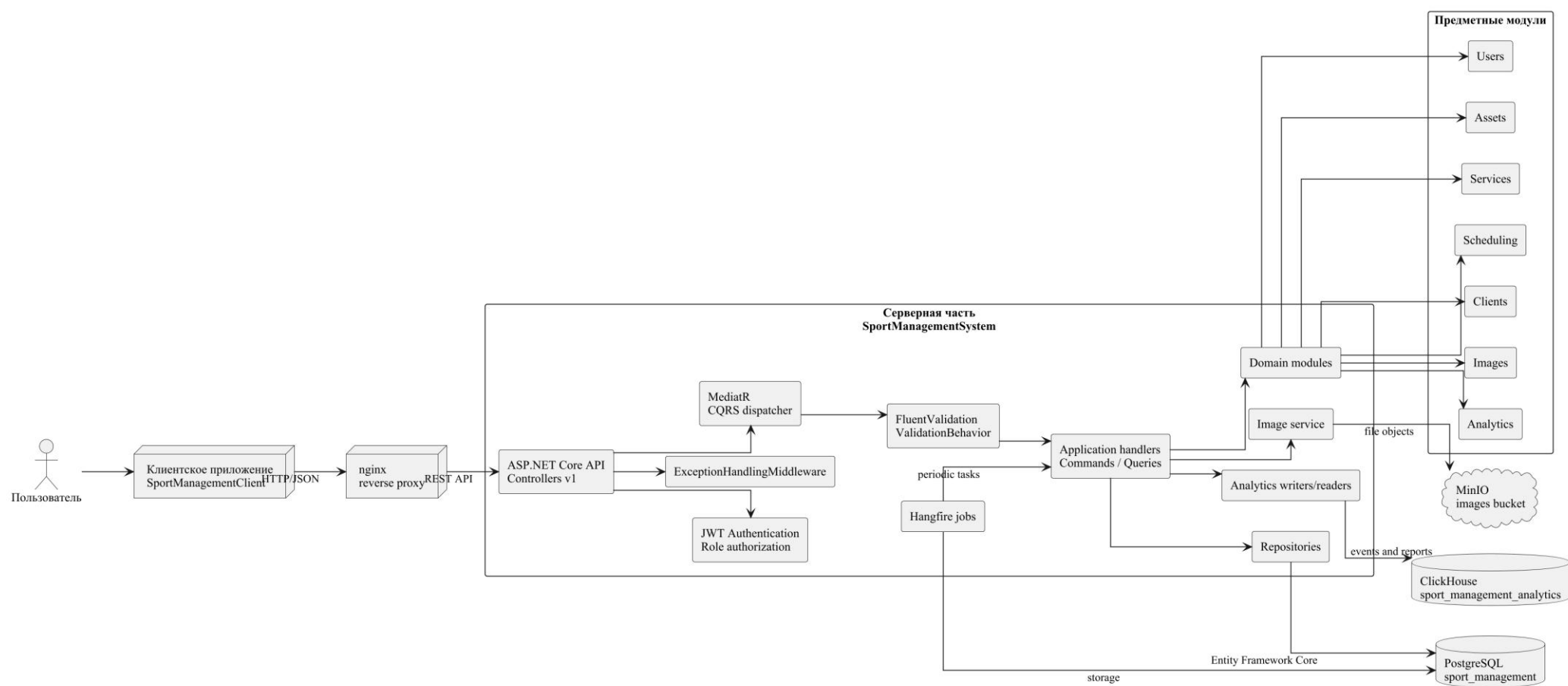


Рисунок Е.5 – Архитектура серверной части информационной системы

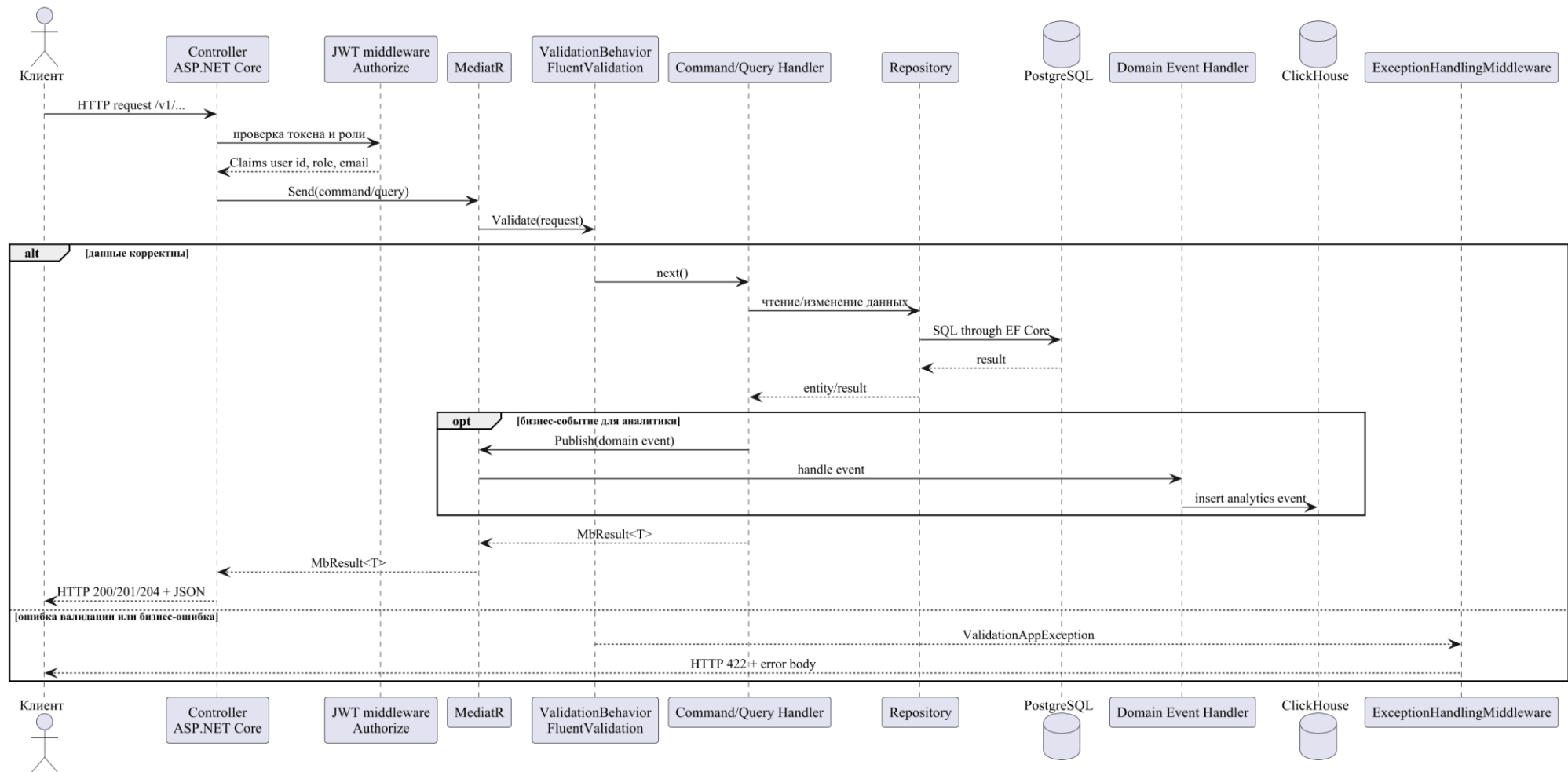


Рисунок Е.6 – Последовательность обработки HTTP-запроса на сервере

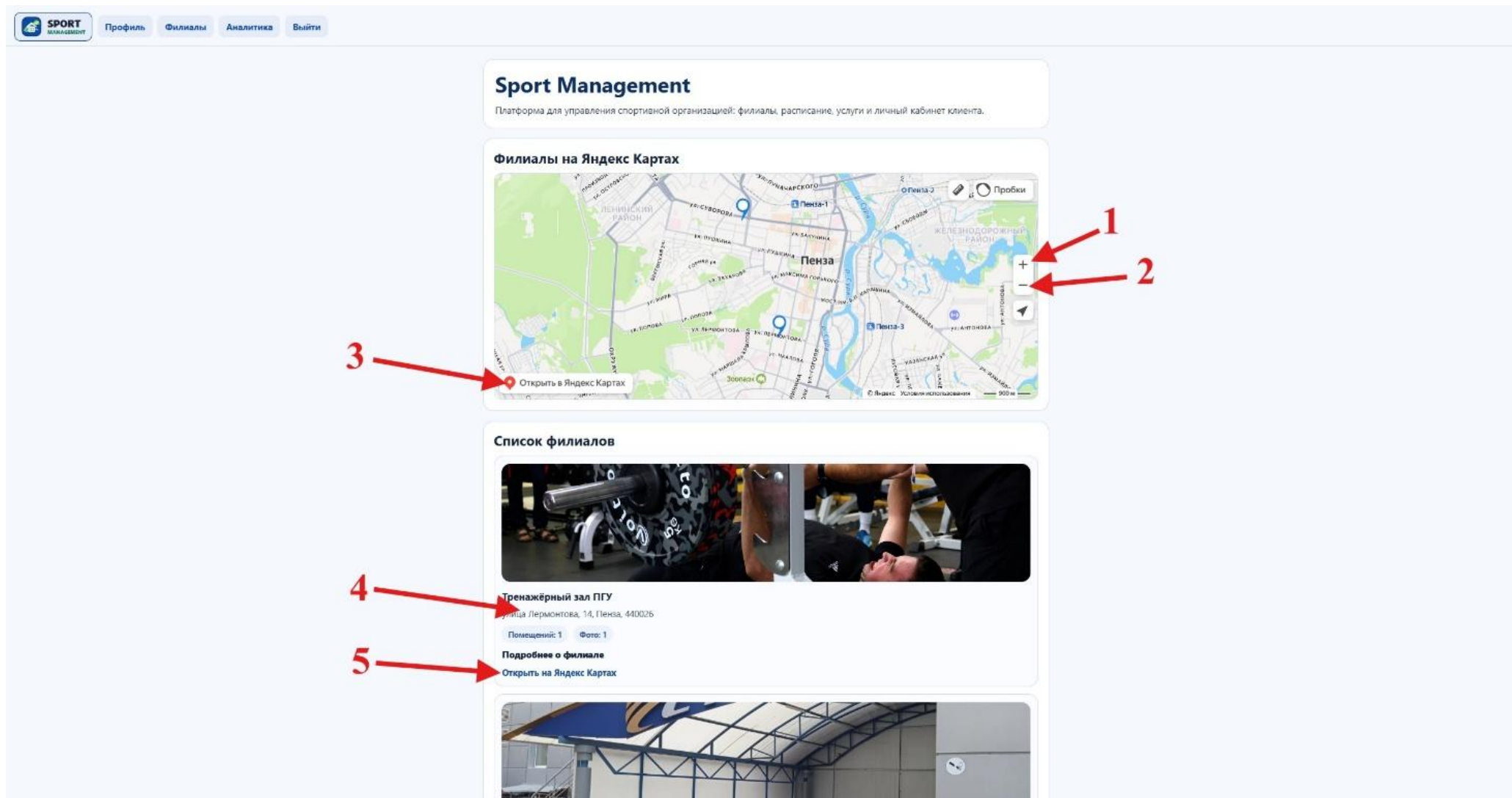


Рисунок Е.7 – Главная страница приложения

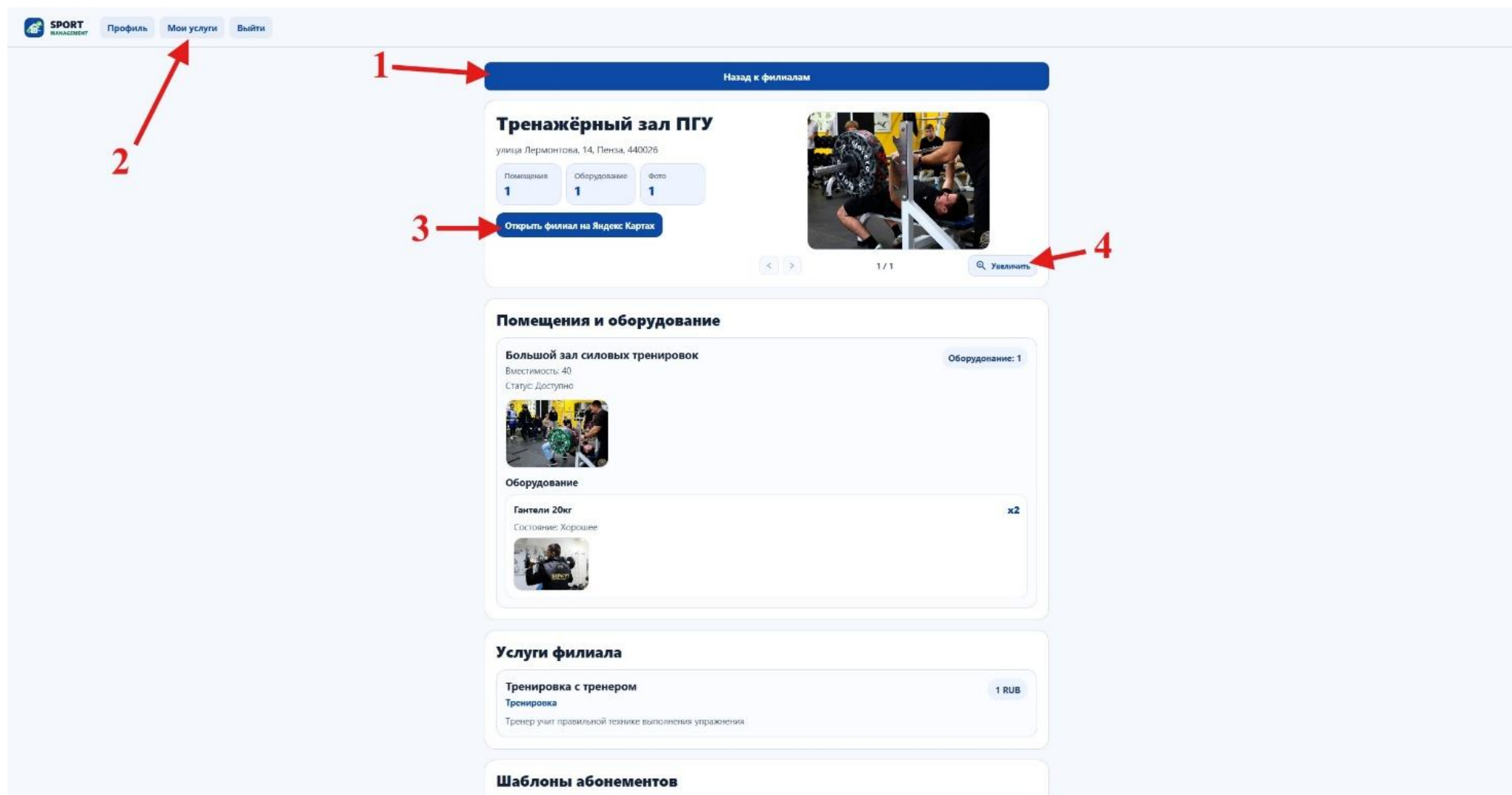


Рисунок Е.8 – Подробная информация о филиале, часть 1

Шаблоны абонементов

Месячный абонемент 1,00 Р

Услуга: Тренировка с тренером

Посещения всех помещений в тренажерном зале

Срок: 30 дн. - Посещений: без ограничений

Купить абонемент

Выбранный шаблон
Месячный абонемент

Услуга
Тренировка с тренером

Параметры
30 дн., посещений: без ограничений

Активация
Абонемент активируется сразу после покупки

Оформить абонемент

Рисунок Е.9 – Подробная информация о филиале, часть 2

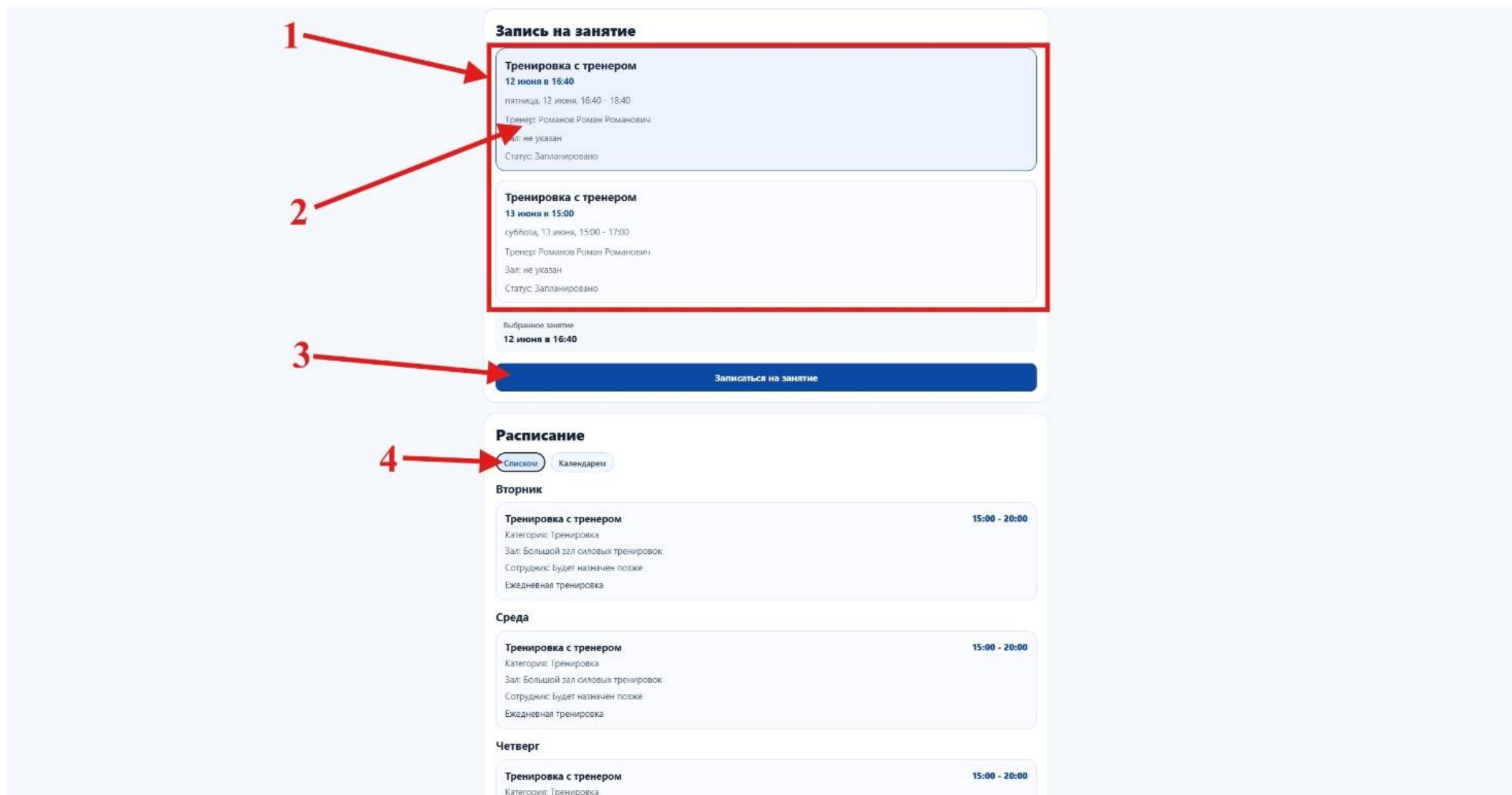


Рисунок Е.10 – Подробная информация о филиале, часть 3

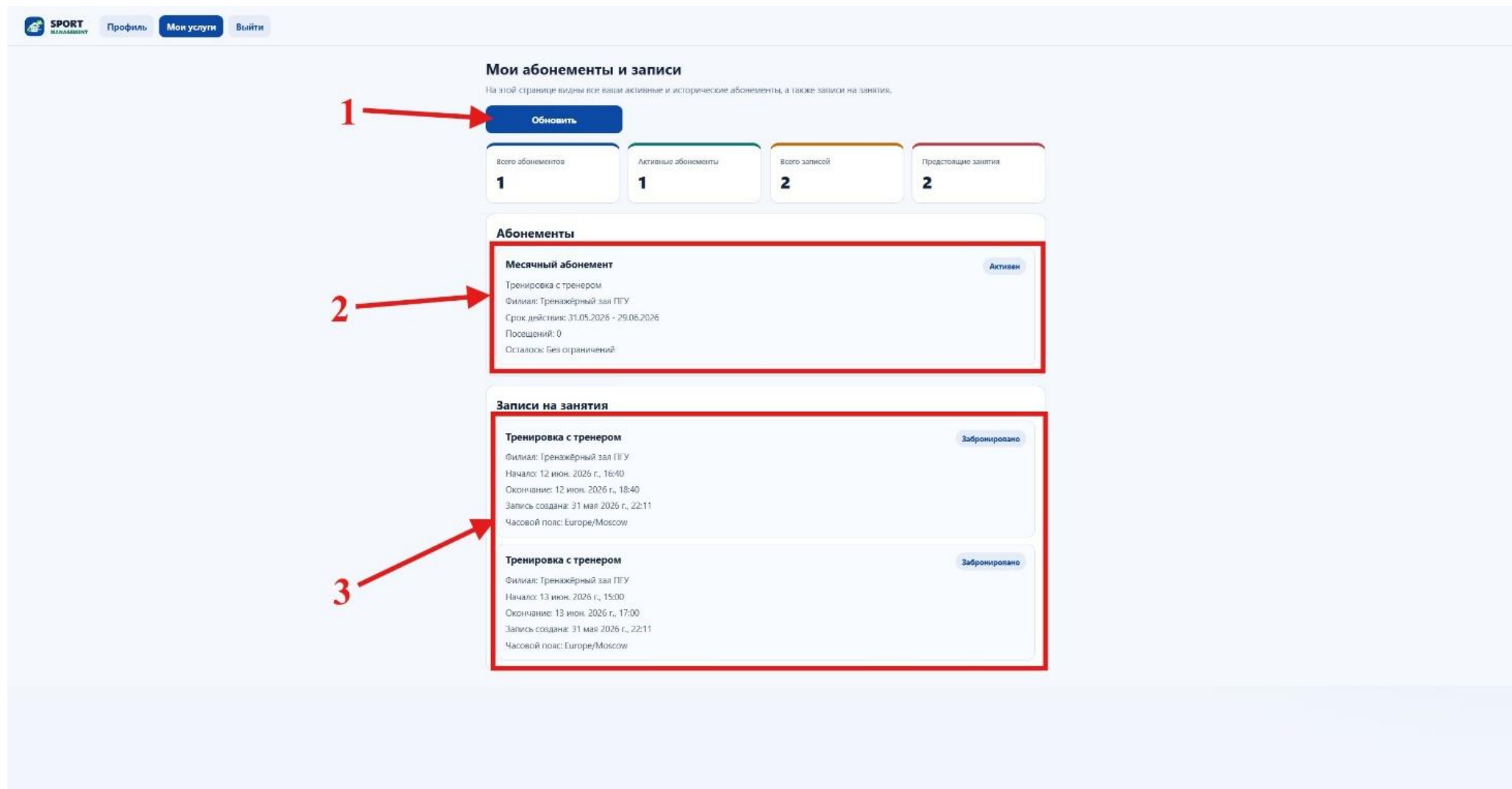


Рисунок Е.11 – Страница абонементов и записей клиента

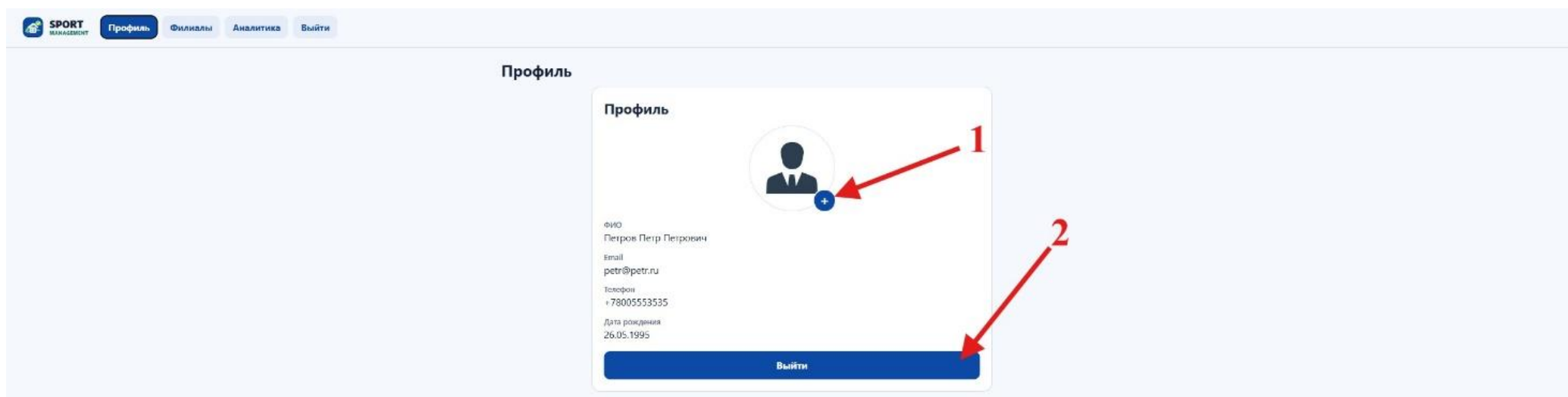


Рисунок Е.12 – Интерфейс управления профилем пользователя

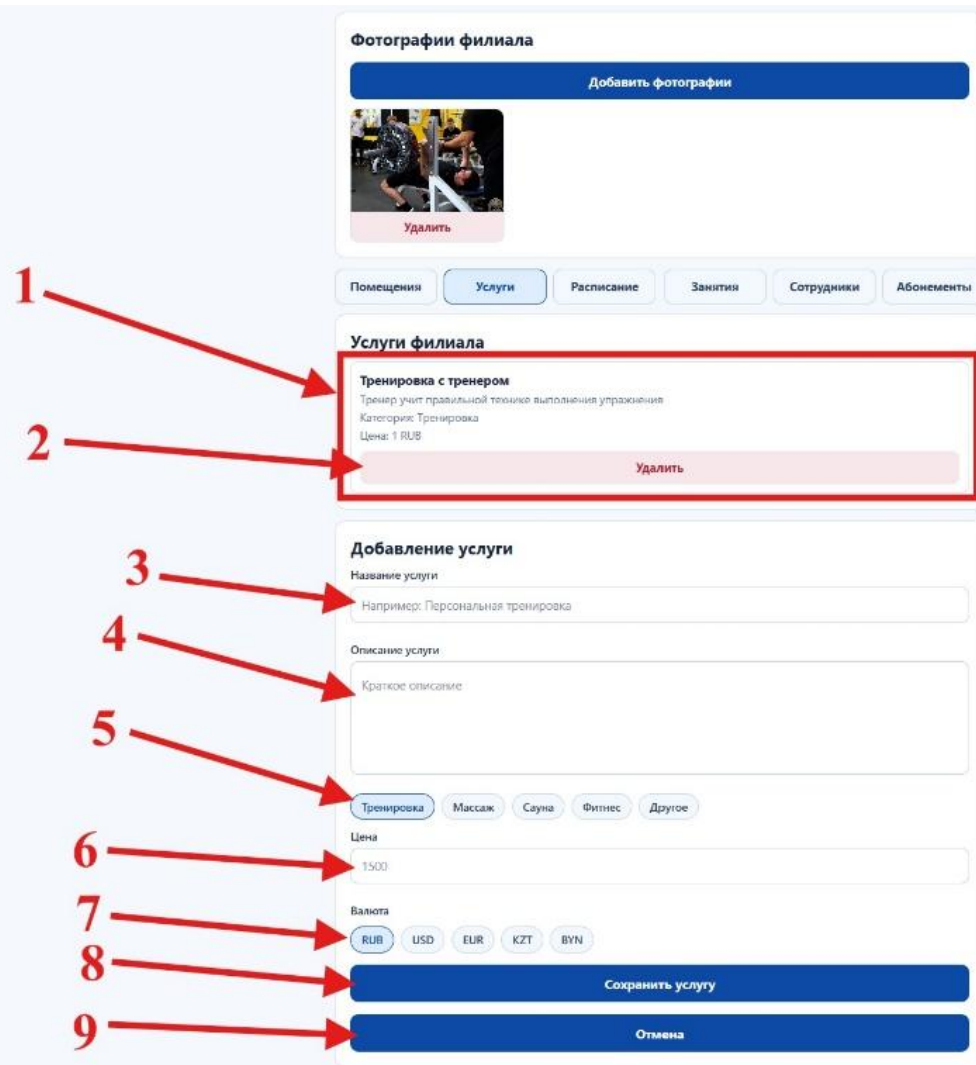


Рисунок Е.13 – Интерфейс управления услугами филиала

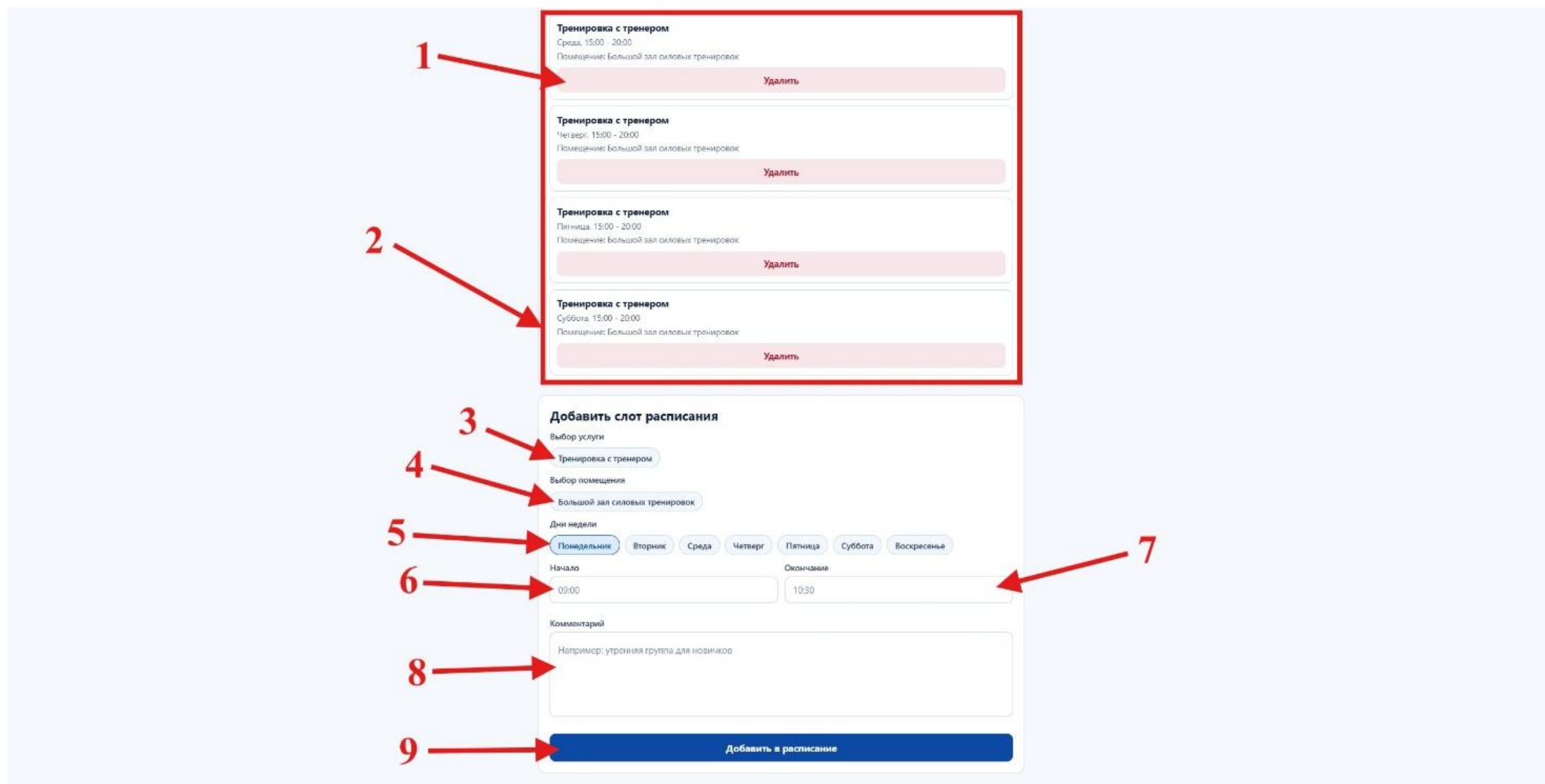


Рисунок Е.14 – Управление расписанием в режиме списка и добавления

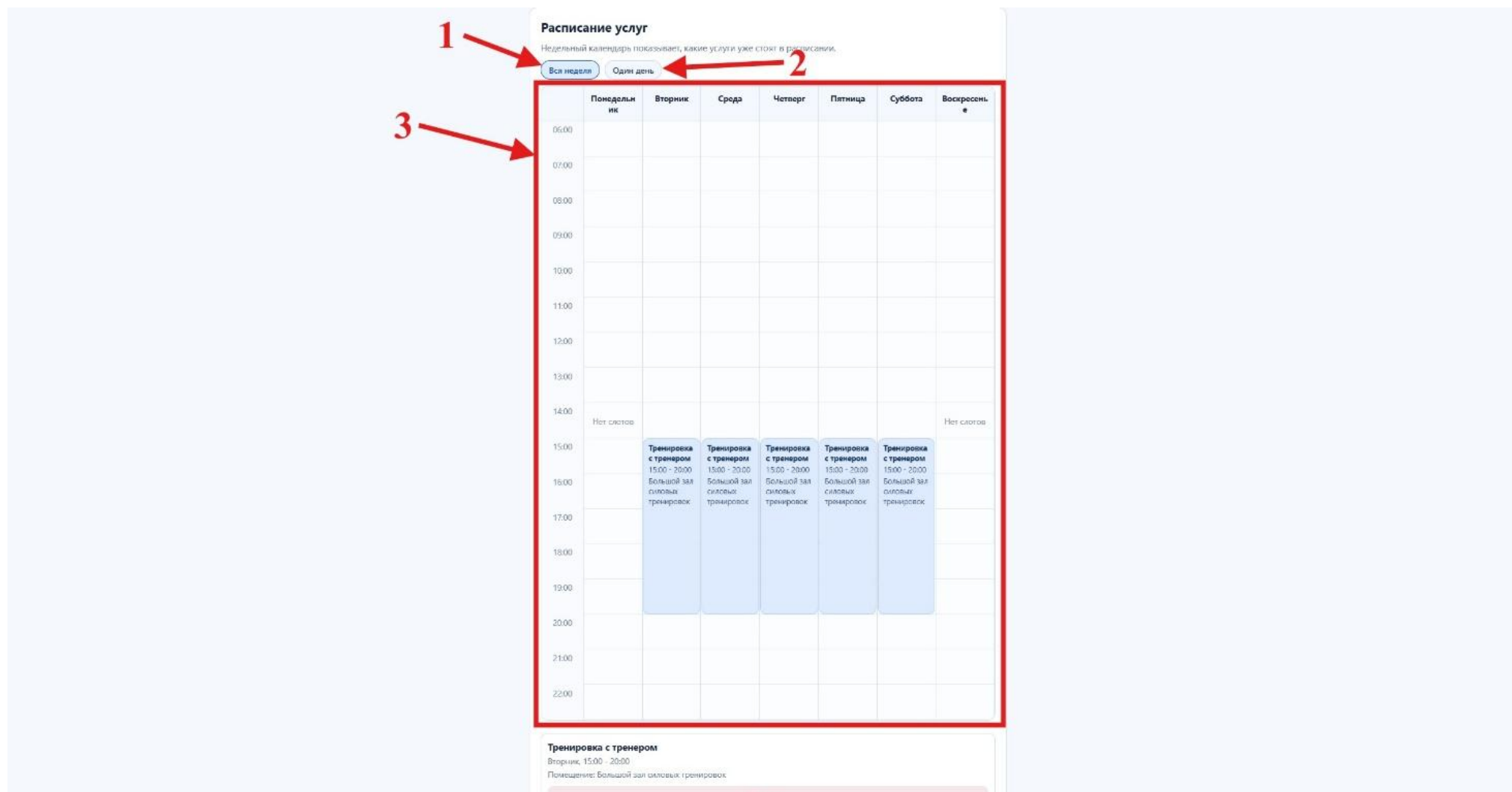


Рисунок Е.15 – Управление расписанием в календарном режиме

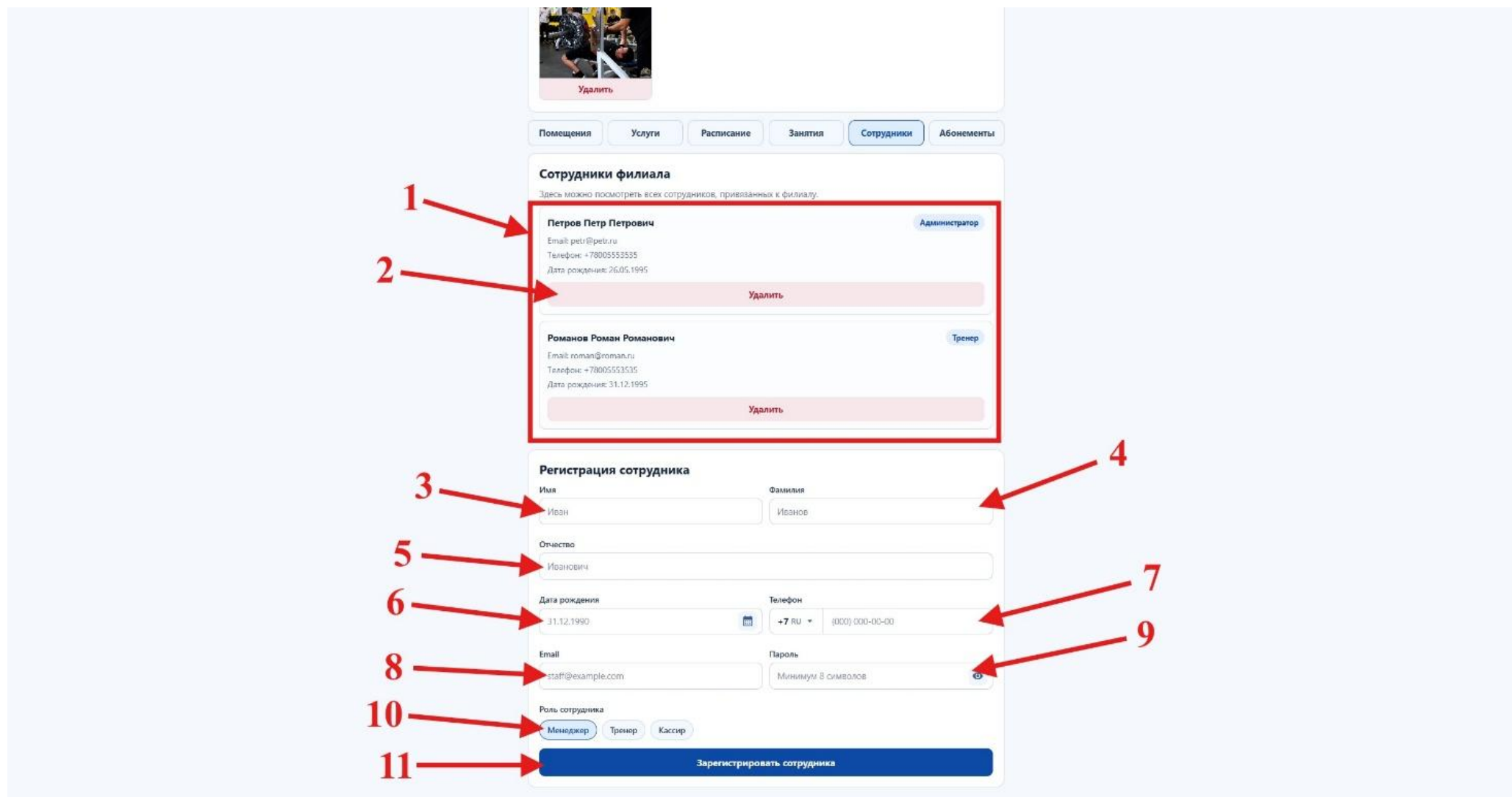


Рисунок Е.16 – Интерфейс управления сотрудниками филиала

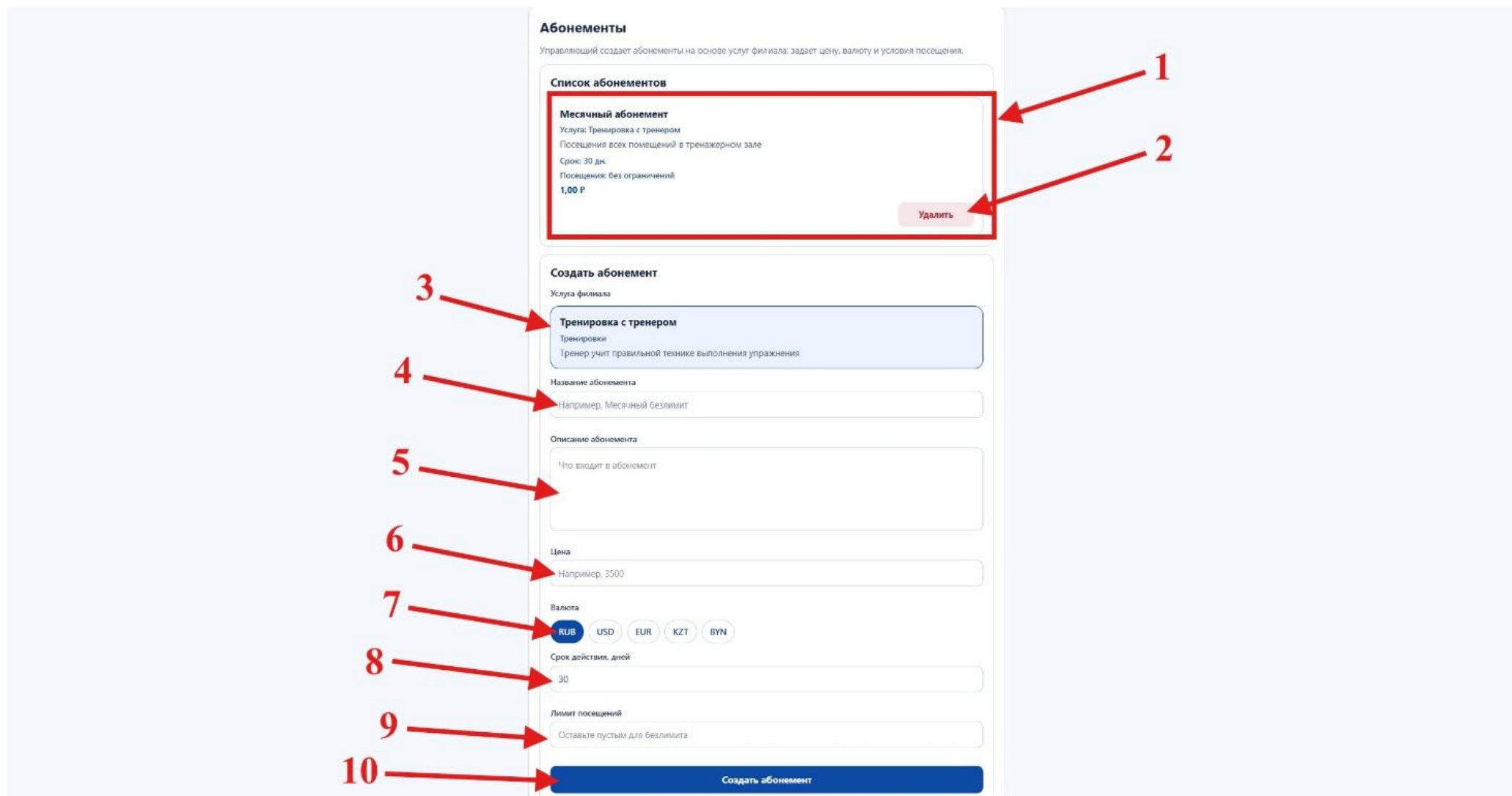


Рисунок Е.17 – Интерфейс управления шаблонами абонементов

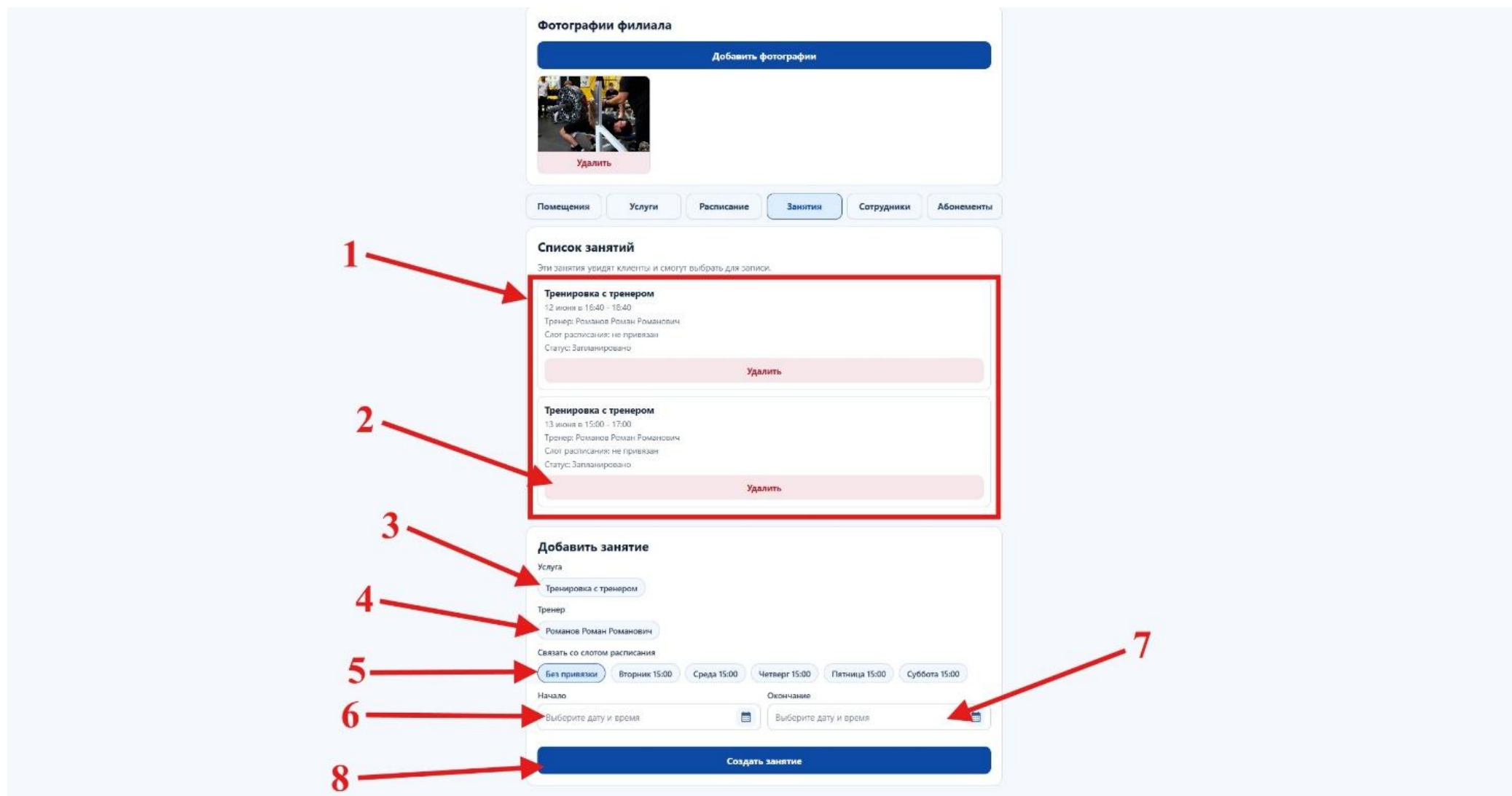


Рисунок Е.18 – Интерфейс управления тренировочными занятиями

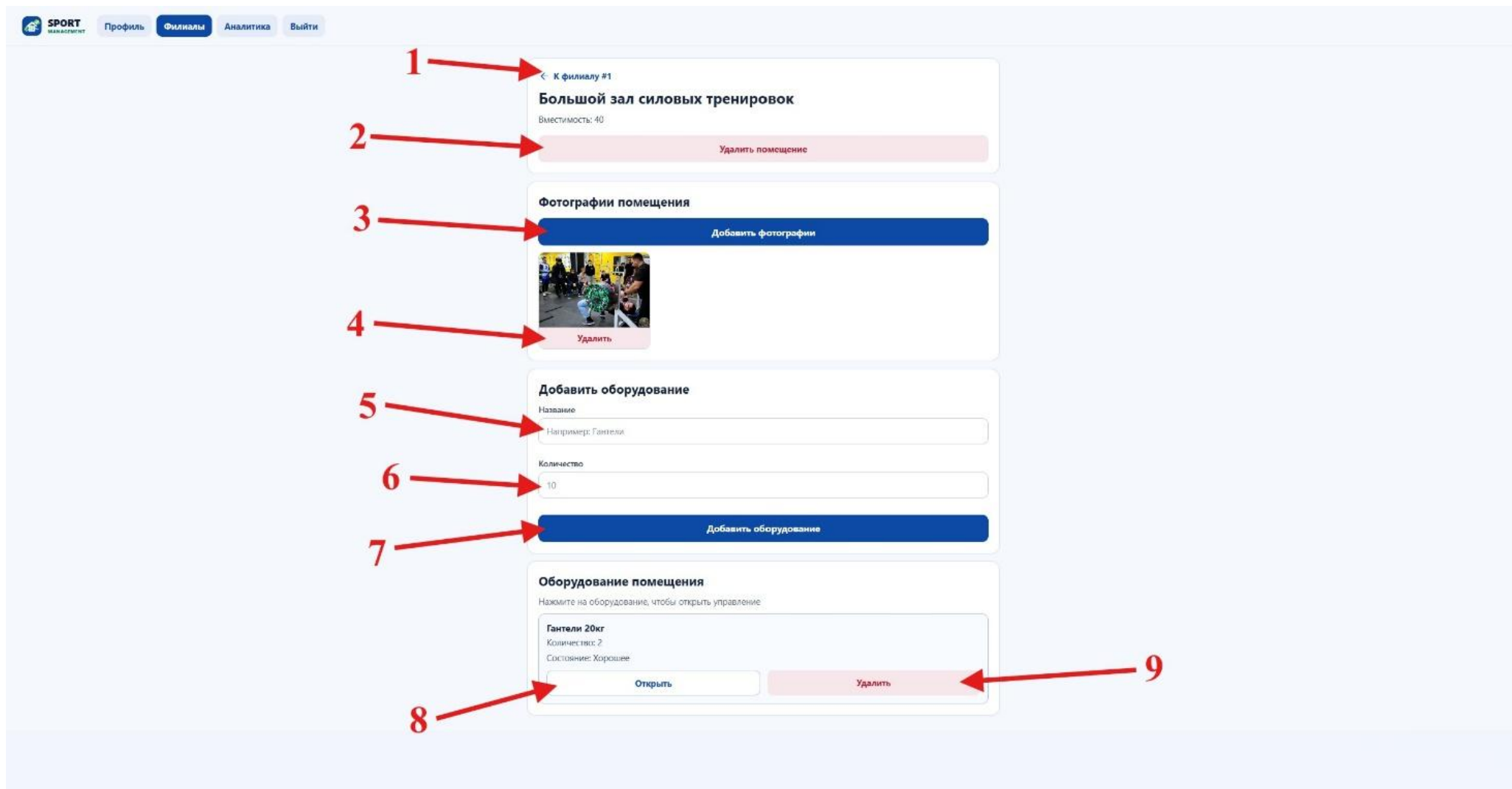


Рисунок Е.19 – Интерфейс управления помещением

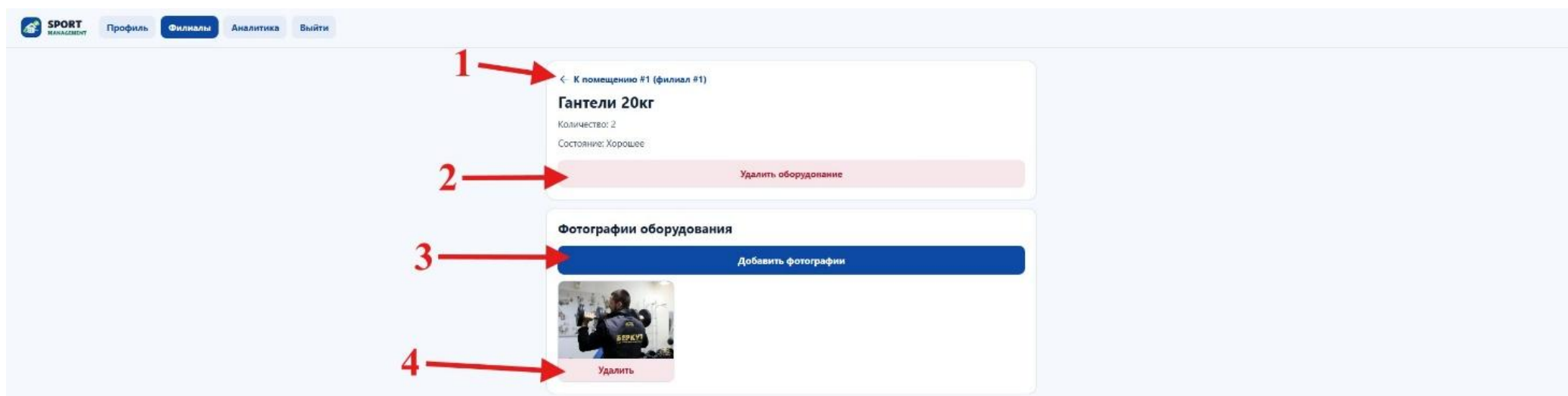


Рисунок Е.20 – Интерфейс управления оборудованием