МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет	*Кафедра*
вычислительной техники	«Вычислительная техника»

Специальность: 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения»

Специализация: «Эксплуатация вычислительных машин, комплексов, систем и сетей специального назначения»

ДИПЛОМНЫЙ ПРОЕКТ

на тему

«WEB-ПРИЛОЖЕНИЕ ДЛЯ ЦЕНТРА ПОДГОТОВКИ КАДРОВ»

Студент 08.06.2026 Седов Сергей Сергеевич
(подпись, дата) (ФИО полностью)

Руководитель 09.06.2026 Слепцов Н.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр 09.06.2026 А.С.Бычков
(подпись, дата) (фамилия, инициалы)

Рецензент зав. каф. «Программирование» ПензГТУ 09.06.26 А.И. Мартышкин
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 22.05.2026 г. № 14)

Заведующий кафедрой _____ М.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 17 июня 2026 г. № 12)

Секретарь ГЭК _____ Е.И. Гурин

Пенза 2026

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

Факультет вычислительной техники

Кафедра «Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____ М.А. Митрохин
«11» 02 2026 г.

ЗАДАНИЕ
НА ДИПЛОМНЫЙ ПРОЕКТ

1. Студент Седов Сергей Сергеевич гр. 21ВОЭ1 факультета Вычислительной техники

2.Тема работы «WEB-приложение для центра подготовки кадров»

Тема утверждена приказом ПГУ № 122/о от "11.02.2026" 2026 г.

3. Руководитель работы к.т.н., доцент кафедры ВТ Слепцов Н.В.

4. Задание на работу (назначение разработки, исходные данные и т.п)
Назначением разработки является создание веб-приложения для автоматизации управления центром подготовки кадров, организации учебного процесса и взаимодействия между участниками системы. Система должна обеспечивать централизованное хранение данных о пользователях, курсах, занятиях, расписании, нормативах и статьях, а также предоставляет интерфейс для трёх категорий пользователей: клиента, тренера и администратора. Исходными данными служат существующие аналоги систем управления обучением, требования к разграничению прав доступа и функциональные требования к автоматизации процессов записи на курсы, составления расписания и ведения результатов клиентов.

5. Перечень подлежащих разработке вопросов
В ходе выполнения дипломного проекта необходимо провести анализ предметной области и существующих аналогов, обосновать актуальность разработки и сформулировать техническое задание; выполнить выбор технологического стека и системы управления базой данных; спроектировать архитектуру приложения, базу данных и пользовательский интерфейс с применением UML-диаграмм. Требуется разработать серверную часть на базе FastAPI с RESTful API, аутентификацией на основе JWT-токенов, системой управления курсами, расписанием и нормативами, а также клиентскую часть на React с компонентной архитектурой, адаптивным интерфейсом и поддержкой трёх ролей пользователей. Необходимо реализовать фоновые задачи с использованием брокера сообщений NATS и обеспечить развёртывание приложения в контейнерах Docker. Кроме того, необходимо обеспечить безопасность хранения персональных данных и провести тестирование системы. В составе пояснительной записки должны быть представлены разделы по безопасности жизнедеятельности и экономическому обоснованию проекта.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1.	Изучение предметной области и анализ существующих аналогов	20	11.02.2026-16.02.2026	
2.	Анализ требований и постановка задачи	20	17.02.2026-20.02.2026	
3.	Выбор технологий проектирования и разработки (Python, FastAPI, SQLAlchemy, PostgreSQL, NATS, Taskiq, TypeScript, React, Docker)	40	21.02.2026-25.02.2026	
4.	Проектирование архитектуры системы, базы данных и пользовательского интерфейса	60	26.02.2026-06.03.2026	
5.	Разработка серверной части (FastAPI, RESTful API, JWT-аутентификация, фоновые задачи)	90	07.03.2026-27.03.2026	
6.	Разработка клиентской части (React, TypeScript, управление состоянием аутентификации, личные кабинеты)	80	01.05.2026-11.05.2026	
7.	Интеграция компонентов, контейнеризация в Docker, настройка Nginx, обеспечение безопасности, тестирование	40	12.05.2026-17.05.2026	
8.	Оформление пояснительной записки (включая БЖД и экономику)	20	18.05.2026-21.05.2026	

Дата выдачи задания "11" 02 2026 г.

Руководитель работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан факультета _____ 16.06.2026 А.Р. Фомина
(подпись, дата)

Реферат

Пояснительная записка: 197 листов, 36 рисунков, 28 таблиц, 18 источников и 3 приложения.

ВЕБ-ПРИЛОЖЕНИЕ, ЦЕНТР ПОДГОТОВКИ КАДРОВ, УПРАВЛЕНИЕ ОБУЧЕНИЕМ, РАСПИСАНИЕ, КУРСЫ, РОЛИ ПОЛЬЗОВАТЕЛЕЙ, КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА, REST API, АУТЕНТИФИКАЦИЯ, JWT, БАЗА ДАННЫХ, PYTHON, FASTAPI, SQLALCHEMY, POSTGRESQL, ALEMBIC, REACT, TYPESCRIPT, DOCKER, NATS, TASKIQ

Цель работы – разработка веб-приложения для автоматизации управления центром подготовки кадров, обеспечивающего организацию учебного процесса, ведение расписания и взаимодействие между клиентами, тренерами и администраторами без необходимости использования сторонних инструментов.

Для достижения поставленной цели было разработано веб-приложение, отвечающее следующим задачам: анализ предметной области и существующих аналогов; обоснование выбора технологий и СУБД; проектирование архитектуры, базы данных и пользовательского интерфейса; реализация серверной и клиентской частей с тестированием; обеспечение безопасности персональных данных и разграничения прав доступа.

В результате разработано веб-приложение, объединяющее управление курсами и активностями, автоматическое формирование персонального расписания клиента при записи на курс, ведение посещаемости и нормативов тренером, администрирование недельного расписания, систему аутентификации на основе JWT-токенов с разграничением доступа для трёх ролей пользователей, а также фоновые задачи на базе брокера сообщений NATS.

Содержание

Введение 6
1 WEB-приложение для центра подготовки кадров 9
1.1 Анализ предметной области 9
1.1.1 Актуальность разработки 9
1.1.2 Анализ существующих аналогов 11
1.1.3 Обоснование актуальности разработки программы 14
1.1.4 Постановка задачи 14
1.2 Выбор технологий и средств разработки приложения 20
1.2.1 Выбор языка программирования 20
1.2.2 Выбор СУБД 22
1.2.3 Графический клиент pgAdmin 4 25
1.2.4 Выбор среды разработки 25
1.2.5 Выбор ORM 27
1.2.6 Выбор веб-фреймворка 30
1.2.7 Выбор очереди задач 32
1.2.8 Выбор брокера сообщений 34
1.2.9 Фреймворк для внедрения зависимостей Dishka 37
1.3 Проектирование информационной системы 38
1.3.1 Архитектура приложения 38
1.3.2 Архитектура серверной части приложения 40
1.3.3 Проектирование базы данных 48
1.4 Разработка приложения 66
1.4.1 Разработка серверной части приложения 66
1.4.2 Разработка клиентской части приложения 93
1.4.3 Тестирование и отладка 113
1.4.4 Развертывание приложения 115
1.4.5 Руководство пользователя 122
2 Охрана труда 132
3 Экономика 135

Заключение 138
Список использованных источников 141
Приложение А – UML-диаграммы 143
Приложение Б – Листинг программы 157
Приложение В – Презентация 197

Перечень сокращений и обозначений

API (Application Programming Interface) – программный интерфейс для взаимодействия между клиентской и серверной частями системы.

Docker – платформа для контейнеризации и развёртывания приложений.

FastAPI – веб-фреймворк на языке Python для разработки серверной части.

HTML (HyperText Markup Language) – язык разметки веб-страниц.

HTTP (HyperText Transfer Protocol) – протокол передачи данных между клиентом и сервером.

HTTPS – защищённая версия протокола HTTP с использованием шифрования.

JSON (JavaScript Object Notation) – текстовый формат обмена данными между клиентом и сервером.

JWT (JSON Web Token) – стандарт защищённой передачи данных в виде подписанного токена, используется для аутентификации пользователей.

NATS – высокопроизводительный брокер сообщений для организации асинхронного взаимодействия между компонентами системы.

ORM (Object-Relational Mapping) – объектно-реляционное отображение, технология связывания объектов программы с таблицами базы данных.

React – библиотека JavaScript для создания пользовательских интерфейсов.

REST (Representational State Transfer) – архитектурный стиль построения веб-сервисов.

SQL (Structured Query Language) – структурированный язык запросов к реляционным базам данных.

SQLAlchemy – библиотека для работы с базами данных на языке Python с поддержкой ORM.

SSH (Secure Shell) – протокол защищённого удалённого доступа к серверу.

SSL (Secure Sockets Layer) – протокол шифрования данных при передаче по сети.

TaskIQ – библиотека для организации фоновых задач на языке Python.

UML (Unified Modeling Language) – унифицированный язык моделирования.

Введение

Эффективность подготовки специалистов во многом определяется не только качеством теоретического обучения, но и уровнем их физической готовности к выполнению профессиональных задач. Современные требования к персоналу, деятельность которого связана с повышенными нагрузками, предполагают системную организацию тренировочного процесса, регулярный контроль показателей и объективную оценку прогресса. В этих условиях физическая подготовка перестаёт быть вспомогательным элементом и становится управляемым процессом, требующим точного учёта и аналитики.

Организация работы центров подготовки кадров включает планирование тренировок, распределение групп, фиксацию результатов нормативов, анализ динамики и формирование отчётности. На практике данные процессы часто осуществляются с использованием бумажных журналов, локальных электронных таблиц или разрозненных файловых хранилищ. Подобный подход формирует зависимость от человеческого фактора, затрудняет обработку накопленных данных и не позволяет оперативно получать сводную информацию о текущем состоянии подготовки.

Особенность физической подготовки заключается в её измеряемости: результаты выражаются в конкретных числовых показателях – времени, количестве повторений, дистанциях, коэффициентах выносливости. При систематическом накоплении такие данные образуют массив, который может быть использован для анализа эффективности тренировочных методик и индивидуального прогресса. Однако без специализированных цифровых инструментов потенциал этих данных остаётся нераскрытым [1].

Современные веб-технологии предоставляют возможность создания централизованных информационных систем, доступных через стандартный браузер и не требующих установки специализированного программного обеспечения на каждом рабочем месте. Это особенно важно для организаций, где необходим одновременный доступ нескольких пользователей с различными

уровнями прав – тренеров, администраторов и руководителей. Веб-приложение позволяет объединить процессы учёта, планирования и анализа в единую цифровую среду.

Следует отметить, что большинство существующих программных решений ориентировано на коммерческую сферу фитнес-услуг. Такие системы преимущественно сосредоточены на учёте посещаемости, расписания и финансовых операций. В условиях центра подготовки кадров приоритетом являются другие задачи: контроль выполнения нормативов, анализ динамики показателей, формирование индивидуальных планов тренировок и подготовка аналитической отчётности. Таким образом, возникает необходимость разработки специализированного программного продукта, учитывающего специфику организационной структуры и целей подготовки.

Дополнительным фактором актуальности является общая тенденция автоматизации процессов. Электронный документооборот, централизованное хранение данных и автоматизация расчётов становятся стандартом в различных отраслях. Отсутствие цифровых инструментов в системе физической подготовки снижает её управляемость и затрудняет интеграцию с другими информационными ресурсами организации.

Разработка веб-приложения для центра подготовки кадров направлена на формирование единой информационной платформы, обеспечивающей хранение данных о занимающихся, фиксацию результатов тренировок и нормативов, автоматический расчёт оценочных показателей и визуализацию динамики развития. Внедрение такой системы позволит повысить прозрачность процессов, сократить временные затраты сотрудников и обеспечить объективность контроля.

Объектом исследования является процесс организации и управления физической подготовкой в центре подготовки кадров.

Предметом исследования выступают методы и средства проектирования веб-приложений для автоматизации учёта, анализа и планирования тренировочного процесса.

Целью выпускной работы является разработка веб-приложения, обеспечивающего комплексную автоматизацию процессов физической подготовки в центре подготовки кадров.

Для достижения поставленной цели необходимо решить следующие задачи:

– исследование особенностей предметной области и определение аналогов;

– сформировать требования к разрабатываемой системе;

– спроектировать архитектуру программного решения и структуру базы данных;

– реализовать серверную и клиентскую части веб-приложения;

– разработать модули учёта результатов и формирования отчётности;

– провести тестирование системы и оценить её работоспособность.

Практическая значимость работы заключается в создании программного продукта, который может быть внедрён в деятельность центра подготовки кадров для повышения эффективности управления физической подготовкой. Разработанная система позволит систематизировать данные, обеспечить их сохранность, улучшить качество аналитики и создать основу для дальнейшего развития цифровых инструментов управления подготовительным процессом.

1 WEB приложение для центра подготовки кадров

1.1 Анализ предметной области

1.1.1 Актуальность разработки

Физическая подготовка персонала, выполняющего задачи повышенной сложности и ответственности, является неотъемлемым элементом системы профессионального обучения. Уровень силы, выносливости, координации, скоростных и функциональных качеств напрямую влияет на способность специалистов эффективно действовать в условиях интенсивных нагрузок. В современных условиях возрастает роль системного подхода к организации тренировочного процесса, основанного на объективных данных, регулярном мониторинге и аналитике результатов.

Исторически контроль физической подготовки осуществлялся преимущественно традиционными методами: ведением бумажных журналов, разрозненных ведомостей, использованием локальных таблиц и ручных расчётов. Такой подход был оправдан в условиях ограниченных объёмов данных и небольшого количества обучаемых. Однако с ростом числа центров подготовки, усложнением программ обучения и повышением требований к качеству контроля данные методы перестают обеспечивать необходимый уровень точности и оперативности. Ручной ввод информации увеличивает вероятность ошибок, затрудняет анализ динамики показателей и усложняет формирование сводной отчётности [2].

Современная образовательная и тренировочная среда характеризуется цифровизацией процессов управления. В различных сферах активно внедряются информационные системы, позволяющие централизованно хранить данные, автоматизировать расчёты и обеспечивать доступ к аналитике в режиме реального времени. Однако в области физической подготовки кадров до сих пор широко распространены частично автоматизированные или полностью ручные методы учёта. Это приводит к фрагментарности информации, отсутствию единой базы данных и затруднениям при принятии управленческих решений.

Дополнительную сложность представляет необходимость индивидуального подхода к каждому обучаемому. Современные методики подготовки предполагают адаптацию нагрузок с учётом текущего уровня физического развития, состояния здоровья и динамики прогресса. Без автоматизированной системы такие корректировки требуют значительных временных затрат со стороны инструкторов и часто выполняются на основе субъективной оценки. Отсутствие инструментов для системного анализа накопленных данных снижает эффективность тренировочного процесса и затрудняет выявление слабых мест в подготовке.

Существующие программные решения, ориентированные на фитнес-индустрию, в основном предназначены для коммерческих организаций и сосредоточены на учёте посещаемости, расписании занятий и финансовых операциях. Они не учитывают специфику подготовки кадров, где приоритетом является контроль нормативов, динамика показателей и формирование индивидуальных программ развития. Кроме того, такие системы редко предусматривают гибкие механизмы аналитики и автоматизированного формирования отчётов для руководства.

В условиях повышения требований к качеству подготовки специалистов возрастает потребность в создании специализированного веб-приложения, способного обеспечить комплексную автоматизацию процессов учёта и анализа физической подготовки. Использование современных веб-технологий позволяет реализовать централизованную систему хранения данных, обеспечить разграничение прав доступа, а также предоставить удобный интерфейс для инструкторов и администраторов.

Автоматизация открывает возможности для внедрения инструментов визуализации динамики показателей, автоматического расчёта итоговых оценок и формирования персонализированных рекомендаций по корректировке тренировочных планов. Это способствует повышению объективности контроля, снижению влияния человеческого фактора и оптимизации временных затрат сотрудников центра подготовки [3].

Таким образом, разработка веб-приложения для центра подготовки кадров, ориентированного на автоматизацию процессов физической подготовки, является актуальной задачей. Создание такой системы позволит повысить эффективность управления тренировочным процессом, обеспечить прозрачность и структурированность данных, а также сформировать основу для дальнейшего развития цифровых инструментов в сфере подготовки специалистов.

1.1.2 Анализ существующих аналогов

Анализ существующих аналогов является важным этапом проектирования программного продукта, поскольку позволяет оценить текущее состояние цифровых решений в рассматриваемой области, выявить их функциональные особенности, ограничения и направления дальнейшего развития. При разработке веб-приложения для центра подготовки кадров особое значение приобретает изучение систем, применяемых для учёта и контроля физической подготовки, а также программных продуктов, используемых в сфере спортивного и тренировочного менеджмента. Такой анализ позволяет определить, какие задачи уже решены существующими инструментами, а какие аспекты требуют доработки или специализированного подхода.

Прямых аналогов разрабатываемой системы не существует. Рассматриваемые решения относятся либо к системам управления фитнес-центрами, либо к организациям, занимающимся военной подготовкой. Несмотря на различие в области применения, анализ их функциональных возможностей позволяет определить существующие подходы к организации подобных информационных систем.

Ниже приведён краткий обзор наиболее распространённых решений с анализом их основных функциональных возможностей и выявлением недостатков, ограничивающих их использование в рамках рассматриваемой предметной области.

– alexfitness.ru (рисунок 1)

Предоставляет: возможность записаться на занятие через приложение, различные виды тренировок и специальных программ, различные абонементы,

просмотр расписания через приложение и сайт, бесплатные пробные занятия и программы питания.

Минус: отсутствие контроля показателей и их динамики, нет возможности сформировать отчет по каждому клиенту для их оценки и составления индивидуального плана.

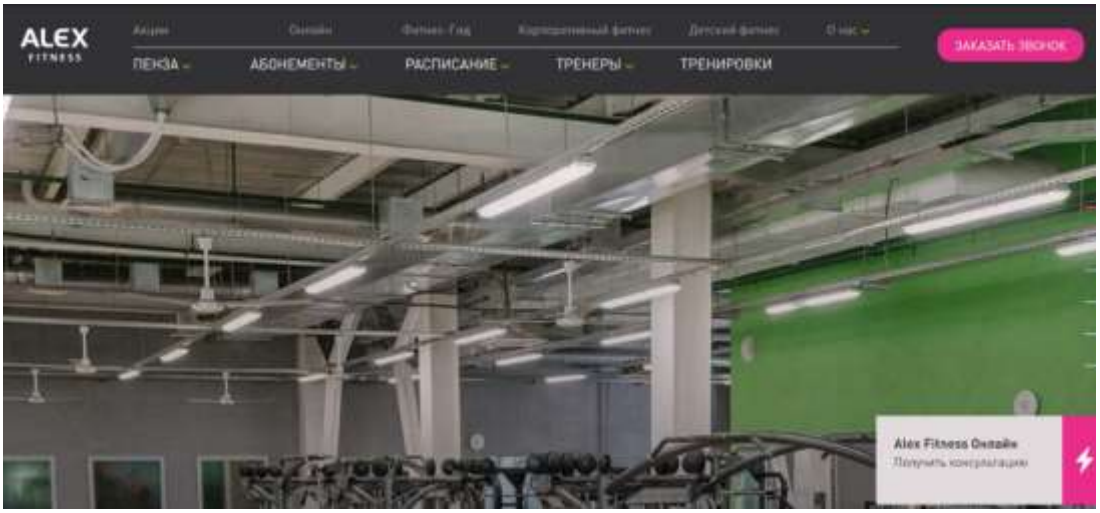


Рисунок 1 – alexfitness.ru

–утцзащитник.рф (рисунок 2)

Предоставляет: различные курсы, опытные преподаватели, учебно-тренировочные центры в разных городах, приближенные к реальности сценарии, тренировки на тактических площадках.

Минус: отсутствие личного кабинет, нет возможности онлайн записаться и посмотреть свое расписание, нет упоминаний об индивидуальном подходе к клиенту, нет упоминаний о возможности сформировать отчет по каждому клиенту для их оценки и составления индивидуального плана.

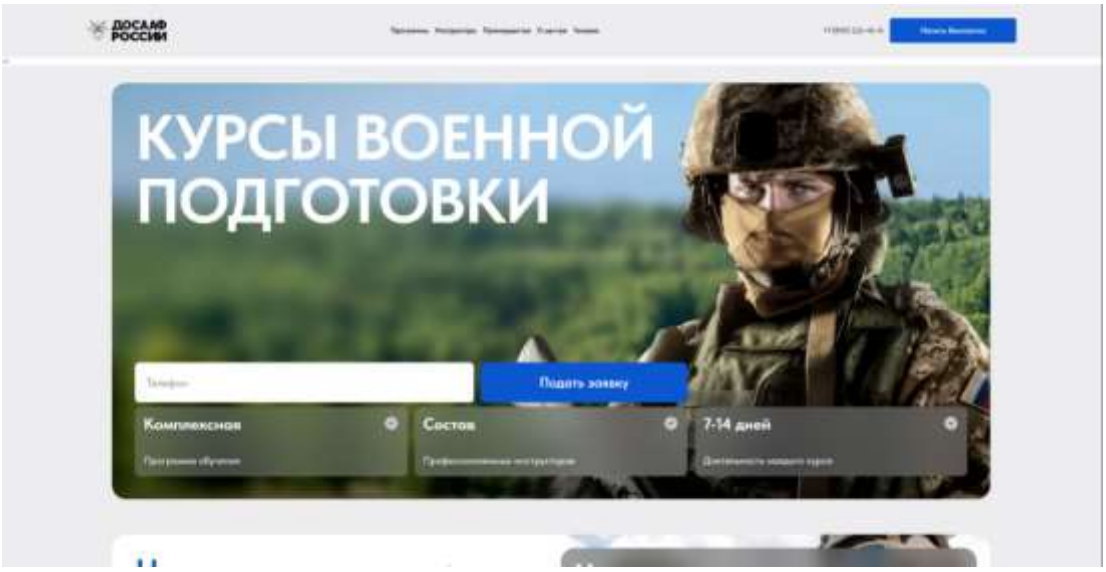


Рисунок 2 – утцзащитник.рф

- uni-gym.ru (рисунок 3)

Предоставляет: возможность просмотреть расписание, несколько филиалов, информацию о каждом тренере, бесплатное пробное занятие.

Минус: отсутствие личного кабинет, нет возможности онлайн записаться, нет упоминаний об индивидуальном подходе к клиенту, нет упоминаний о возможности сформировать отчет по каждому клиенту для их оценки.

Все перечисленные сервисы не являются прямыми аналогами разрабатываемой системы, а также большинство из них не имеют возможности онлайн записи на занятие, не имеют возможности просмотреть динамику показателей каждого клиента и формировать по ним отчет.

Рисунок 3 – uni-gym.ru

1.1.3 Обоснование актуальности разработки программы

Разрабатываемое веб-приложение для центра подготовки кадров направлено на комплексную автоматизацию процессов организации и контроля физической подготовки. В отличие от большинства существующих решений, ориентированных преимущественно на коммерческие фитнес-центры, проектируемая система учитывает специфику смешанной модели работы, включающей как платные занятия, так и бесплатные программы, финансируемые за счёт внутренних или государственных источников для отдельных категорий граждан.

Одним из ключевых преимуществ разрабатываемой системы является объединение в рамках единой платформы управленческих, аналитических и организационных инструментов. Это позволяет исключить фрагментарность учёта и обеспечить централизованное хранение данных о клиентах, тренировках и результатах.

Система решает ряд проблем, выявленных при анализе существующих аналогов:

– отсутствие комплексного модуля аналитики динамики показателей;
– ориентация аналогов преимущественно на финансовый учёт;
– ограниченная поддержка индивидуальных программ;
– отсутствие централизованных уведомлений;
– отсутствие возможности сформировать отчет по каждому клиенту.

1.1.4 Постановка задачи

Требуется разработать веб-приложение для центра подготовки кадров, обеспечивающее автоматизацию процессов организации, контроля и анализа физической подготовки клиентов.

Система должна соответствовать следующим функциональным требованиям.

Модуль регистрации и управления пользователями:

– регистрация новых пользователей (клиентов, тренеров, администраторов);
– авторизация и аутентификация в системе;

– разграничение прав доступа в зависимости от роли;

– хранение персональных данных в защищённом виде;

– восстановление доступа к учётной записи.

Модуль занятий:

– создание и редактирование расписания тренировок;

– разделение занятий на платные и бесплатные программы;

– возможность записи клиентов на выбранные занятия;

– формирование индивидуальных тренировочных планов с учётом целей клиента;

– ведение истории посещений.

Модуль учёта показателей:

– фиксация результатов выполнения нормативов и контрольных упражнений;

– сохранение показателей с привязкой к дате;

– отображение динамики изменений в личном кабинете.

Личные кабинеты:

– кабинет клиента с доступом к расписанию, истории занятий и показателей;

– кабинет тренера с возможностью управления группами, внесения результатов и формирования планов;

– кабинет администратора с функциями управления пользователями, программами и отчётностью.

Модуль уведомлений:

– напоминания о предстоящих занятиях;

– уведомления об изменениях расписания;

– информирование о публикации новых результатов или отчётов. Разрабатываемая автоматизированная система должна соответствовать следующим техническим требованиям:

– тип приложения: клиент-серверное веб-приложение с адаптивным интерфейсом, доступное через современные веб-браузеры на различных устройствах (персональные компьютеры, планшеты, смартфоны).

– клиентская часть: Next.js (React);

– серверная часть: Python;

– база данных: PostgreSQL.

– интерфейс должен быть интуитивно понятным, обеспечивать удобную навигацию и корректное отображение на устройствах различного типа.

– производительность системы должна обеспечивать отклик основных операций (авторизация, загрузка данных, формирование отчётов) в пределах допустимых временных значений.

– система должна обеспечивать круглосуточный доступ при наличии подключения к сети Интернет.

Требования безопасности:

– хранение паролей в хешированном виде;

– разграничение прав доступа;

– защита персональных данных в соответствии с действующим законодательством Российской Федерации.

Минимальные требования к серверному оборудованию:

– процессор: 2 ядра;

– оперативная память: не менее 4 ГБ;

– достаточный объём дискового пространства для хранения базы данных и резервных копий.

Разрабатываемая система должна быть готова к опытной эксплуатации, пройти функциональное тестирование и обеспечить решение задач, поставленных в рамках выпускной квалификационной работы. На рисунках 4-6 представлены диаграммы вариантов использования для разных пользователей приложения.

Рисунок 4 –Диаграмма вариантов использования приложения (Клиент)

Рисунок 5 –Диаграмма вариантов использования приложения (Тренер)

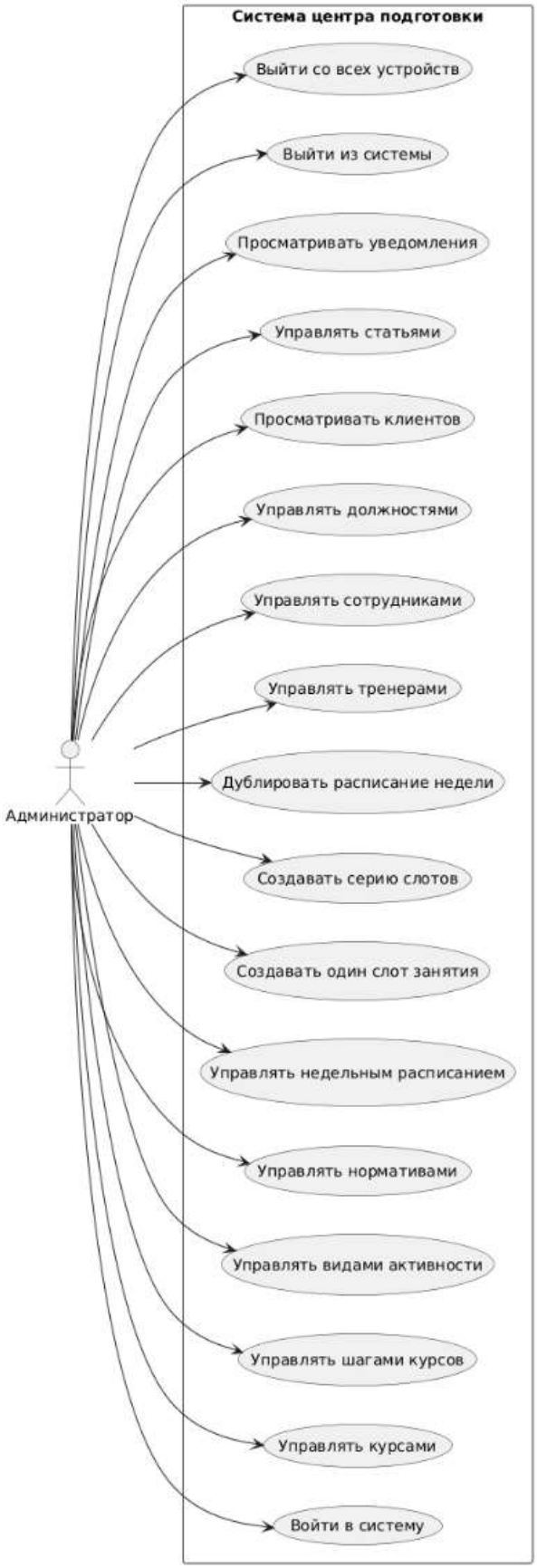


Рисунок 6 –Диаграмма вариантов использования приложения (Администратор)

1.2 Выбор технологий и средств разработки приложения

В данном разделе будут описаны технологии и средства разработки, использованные при разработке WEB приложения для центра подготовки кадров.

Ниже приведен список выбранных технологий и средств разработки:

– Python;

– FastAPI;

– Taskiq;

– NATS;

– Dishka;

– SQLAlchemy;

– PgAdmin4;

– PyCharm;

– WebStorm;

– JavaScript.

1.2.1 Выбор языка программирования

Язык программирования Java.

Java – это объектно-ориентированный язык программирования, разработанный для создания надёжных и кроссплатформенных приложений. Программы на Java выполняются на виртуальной машине JVM, что обеспечивает возможность запуска приложений на различных операционных системах без изменения исходного кода.

Java обладает строгой типизацией, развитой системой управления памятью и механизмом сборки мусора. Для разработки серверных приложений часто используются фреймворки Spring и Spring Boot, позволяющие создавать масштабируемые веб-сервисы. Однако разработка на Java обычно требует большего объёма кода и более сложной настройки проекта.

Язык программирования Go.

Go – компилируемый язык программирования, разработанный компанией Google для создания высокопроизводительных серверных приложений. Он

отличается высокой скоростью выполнения программ, простым синтаксисом и встроенной поддержкой многопоточности.

Go часто используется для разработки сетевых сервисов и микросервисных архитектур. Преимуществами языка являются высокая производительность и эффективная работа с параллельными процессами. К недостаткам можно отнести меньшую распространённость и менее развитую экосистему веб-фреймворков по сравнению с другими языками.

Язык программирования Python.

Python – это высокоуровневый интерпретируемый язык программирования с динамической типизацией. Данный язык отличается простым и понятным синтаксисом, что позволяет ускорить процесс разработки и облегчает сопровождение программного кода. Python поддерживает несколько парадигм программирования, включая объектно-ориентированное и функциональное программирование, а также обладает большим количеством библиотек для разработки веб-приложений. Плюсы и минусы языков программирования приведены в таблице 1.

Таблица 1 – плюсы и минусы языков программирования

Характеристика языка	Go	Java	Python
Развитая экосистема	+-	+	+
Высокая производительность	+	+	-
Встроенная удобная поддержка многопоточности	+	+	+-
Быстрая разработка веб-приложений	+	-	+
Простота сопровождения кода	+	-	+
Кроссплатформенность	+	+	+

Для создания серверной части веб-приложений на Python используются современные фреймворки, такие как FastAPI, Flask и Django, позволяющие реализовывать REST API, обрабатывать HTTP-запросы и взаимодействовать с базами данных. К недостаткам языка можно отнести более низкую производительность по сравнению с компилируемыми языками, однако для большинства информационных систем данный недостаток не является критичным [4].

Язык программирования JavaScript

JavaScript – это язык программирования, широко применяемый для разработки клиентской части веб-приложений. Он выполняется непосредственно в веб-браузере и обеспечивает интерактивность пользовательского интерфейса. JavaScript позволяет создавать динамические веб-страницы, обрабатывать действия пользователя и взаимодействовать с серверной частью приложения через HTTP-запросы.

Преимуществами JavaScript являются высокая распространённость, поддержка всеми современными браузерами и большое количество библиотек и фреймворков для разработки интерфейсов, таких как React, Vue и Angular. К недостаткам можно отнести динамическую типизацию и необходимость учитывать особенности работы различных браузеров.

После сравнения возможностей популярных языков программирования было принято решение использовать Python для разработки серверной части приложения и JavaScript для реализации клиентской части. Python обеспечивает удобную разработку API и взаимодействие с базой данных PostgreSQL, тогда как JavaScript используется для создания интерактивного пользовательского интерфейса веб-приложения.

1.2.2 Выбор СУБД

СУБД PostgreSQL.

В качестве системы управления базами данных была выбрана PostgreSQL – свободно распространяемая объектно-реляционная система управления базами данных, предоставляющая широкий набор возможностей для хранения, обработки

и анализа данных. Данная СУБД распространяется на бесплатной основе и обладает высокой надежностью и расширяемостью.

Среди преимуществ PostgreSQL можно выделить:

– высокую стабильность и надежность хранения данных;

– поддержку сложных запросов и транзакций;

– расширяемость и богатый набор типов данных;

– кроссплатформенность;

К недостаткам PostgreSQL можно отнести более сложную настройку и администрирование по сравнению с некоторыми другими СУБД.

Выбор PostgreSQL обусловлен тем, что данная СУБД хорошо подходит для веб-приложений с большим количеством операций чтения и записи данных. Она обеспечивает надежное хранение информации [5].

СУБД MySQL.

MySQL – одна из самых распространённых реляционных систем управления базами данных, широко используемая при разработке веб-приложений. Она отличается высокой производительностью и простотой использования, что делает её популярным выбором среди разработчиков.

Преимущества MySQL:

– высокая скорость работы при выполнении типовых операций;

– наличие большого количества различных движков хранения данных (InnoDB, MyISAM и др.), предназначенных для решения различных задач;

– простота установки и настройки;

– широкое распространение и большое сообщество пользователей.

Однако MySQL имеет и ряд недостатков. Некоторые версии системы имеют ограничения в реализации сложных транзакций и аналитических операций. Также при работе со сложными запросами и большими объемами данных производительность может снижаться.

СУБД SQLite.

SQLite – компактная встраиваемая система управления базами данных, которая хранит всю базу данных в одном файле. В отличие от клиент-серверных

СУБД, SQLite не требует отдельного сервера и может работать непосредственно внутри приложения.

Преимущества SQLite:

– простота установки и использования;
– отсутствие необходимости в отдельном сервере базы данных;
– высокая скорость работы при небольших объёмах данных;
– компактность и минимальные требования к ресурсам.

К недостаткам SQLite можно отнести ограниченные возможности масштабирования и сложности при одновременной работе большого количества пользователей. Также данная СУБД менее подходит для проектов, где требуется интенсивная работа с данными и высокая нагрузка на систему. Плюсы и минусы разных СУБД приведены в таблице 2.

Таблица 2 – плюсы и минусы разных СУБД

Характеристика СУБД	PostgreSQL	MySQL	SQLite
Реляционная модель данных	+	+	+
Поддержка сложных SQL-запросов	+	+-	-
Полноценная поддержка транзакций	+	+	-
Высокая надежность хранения данных	+	+	-
Поддержка расширенных типов данных	+	-	-
Масштабируемость	+	+	-
Простота установки и настройки	-	+	+
Подходит для высоконагруженных веб-приложений	+	+	-

1.2.3 Графический клиент pgAdmin 4

pgAdmin 4 – это графический инструмент администрирования системы управления базами данных PostgreSQL. Данное приложение распространяется с открытым исходным кодом и предназначено для упрощения работы разработчиков и администраторов с базами данных. pgAdmin предоставляет удобный графический интерфейс, позволяющий выполнять SQL-запросы, управлять объектами базы данных и анализировать её состояние без необходимости постоянного использования командной строки.

К основным преимуществам pgAdmin 4 можно отнести:

– удобный и наглядный интерфейс, позволяющий работать с базой данных в графическом режиме;

– широкий набор инструментов администрирования, включая создание, изменение и удаление объектов базы данных;

– встроенный редактор SQL-запросов с подсветкой синтаксиса и возможностью анализа результатов выполнения;

– поддержку безопасного подключения к серверу PostgreSQL с использованием механизмов аутентификации и шифрования;

– наличие инструментов мониторинга, позволяющих отслеживать состояние базы данных и анализировать производительность запросов.

Использование pgAdmin 4 значительно упрощает процесс разработки и администрирования базы данных PostgreSQL. Благодаря наглядному интерфейсу и встроенным инструментам анализа разработчик может быстрее выполнять необходимые операции с базой данных, проверять корректность запросов и контролировать работу системы.

1.2.4 Выбор среды разработки

Среда разработки PyCharm.

PyCharm – это интегрированная среда разработки, предназначенная для программирования на языке Python и разработанная компанией JetBrains. Она широко используется при создании серверных приложений, веб-сервисов и API. Среда предоставляет разработчику большое количество инструментов,

позволяющих ускорить процесс разработки и повысить качество программного кода.

К основным преимуществам PyCharm можно отнести:

– интеллектуальный редактор кода с автодополнением и подсветкой синтаксиса;

– встроенные инструменты отладки и анализа кода;

– удобную работу с виртуальными окружениями и зависимостями Python;

– интеграцию с системами контроля версий;

– поддержку популярных фреймворков для разработки веб-приложений.

К недостаткам PyCharm можно отнести достаточно высокие требования к ресурсам компьютера. Кроме того, для использования полного набора возможностей среды требуется платная версия.

Среда разработки WebStorm.

WebStorm – интегрированная среда разработки, предназначенная для создания веб-приложений на языках JavaScript и TypeScript. Данная среда также разработана компанией JetBrains и предоставляет широкий набор инструментов для разработки клиентской части веб-приложений.

Преимущества WebStorm:

– интеллектуальное автодополнение кода и подсветка синтаксиса;

– поддержка современных веб-технологий (HTML, CSS, JavaScript, TypeScript);

– встроенные средства отладки и тестирования веб-приложений;

– удобная навигация по структуре проекта;

– интеграция с системами контроля версий и менеджерами пакетов.

Недостатками WebStorm являются повышенные требования к аппаратным ресурсам компьютера и необходимость приобретения лицензии для использования всех возможностей среды.

Среда разработки VS Code.

Visual Studio Code – это бесплатный редактор исходного кода, разработанный компанией Microsoft и поддерживающий большое количество языков программирования. Благодаря системе расширений VS Code может использоваться как полноценная среда разработки для различных технологий, включая Python, JavaScript.

Преимущества VS Code:

– бесплатное распространение и открытая экосистема расширений;

– высокая скорость работы и сравнительно небольшие требования к ресурсам компьютера;

– поддержка большого количества языков программирования;

– возможность расширения функциональности с помощью плагинов;

– встроенная поддержка систем контроля версий.

К недостаткам VS Code можно отнести необходимость самостоятельной настройки среды разработки и установки дополнительных расширений для полноценной работы.

После анализа возможностей различных сред разработки было принято решение использовать PyCharm для разработки серверной части приложения на языке Python и WebStorm для реализации клиентской части веб-приложения. Данные среды разработки обеспечивают удобные инструменты для работы с соответствующими технологиями и позволяют повысить эффективность процесса разработки.

1.2.5 Выбор ORM

SQLAlchemy

SQLAlchemy – это популярная библиотека для работы с базами данных на языке программирования Python, реализующая механизм объектно-реляционного отображения. Данная библиотека позволяет взаимодействовать с базой данных с помощью объектов Python, что значительно упрощает разработку приложений и уменьшает необходимость написания большого количества SQL-запросов.

К основным преимуществам SQLAlchemy можно отнести:

– гибкую систему работы с базами данных и поддержку различных СУБД;

– возможность использования как ORM-подхода, так и написания SQL-запросов;

– высокую производительность и оптимизацию работы с данными;

– развитую экосистему и активное сообщество разработчиков;

– хорошую интеграцию с современными Python-фреймворками для разработки веб-приложений.

К недостаткам SQLAlchemy можно отнести относительно высокий порог входа для начинающих разработчиков, а также необходимость более детальной настройки моделей и соединений с базой данных.

Django ORM

Django ORM – это встроенная система объектно-реляционного отображения, входящая в состав веб-фреймворка Django. Она позволяет разработчикам работать с базой данных через Python-объекты и автоматически генерирует SQL-запросы для выполнения операций чтения и записи данных.

Преимущества:

– простота использования и удобный синтаксис запросов;

– тесная интеграция с фреймворком Django;

– автоматическое создание и управление миграциями базы данных;

– большое количество готовых инструментов для работы с моделями.

Однако использование Django ORM связано с определёнными ограничениями. Данная система тесно привязана к фреймворку Django и менее гибка при использовании в проектах, где применяется другой технологический стек.

Peewee

Peewee – это лёгкая ORM-библиотека для Python, предназначенная для упрощения работы с реляционными базами данных. Она обладает относительно простым синтаксисом и может использоваться в небольших проектах или приложениях с несложной структурой данных.

Преимущества:

– компактность и простота использования;
– небольшое количество зависимостей;
– удобный и понятный интерфейс работы с моделями данных;
– поддержка популярных СУБД.

К недостаткам Peewee можно отнести меньшую функциональность по сравнению с более крупными ORM-решениями. Также библиотека имеет менее развитую экосистему и используется реже в крупных проектах.

После анализа возможностей различных ORM-библиотек для реализации взаимодействия приложения с базой данных была выбрана библиотека SQLAlchemy. Она обеспечивает высокую гибкость работы с базами данных, поддержку различных СУБД, а также возможность эффективного построения сложных запросов. Использование SQLAlchemy позволяет удобно реализовать слой доступа к данным в разрабатываемом веб-приложении и обеспечить надёжную работу системы с базой данных PostgreSQL [6]. Сравнение разных ORM приведено в таблице 3.

Таблица 3 – плюсы и минусы разных ORM

Характеристики ORM	SQLAlchemy	Django ORM	Peewee
Возможность писать сложные SQL-запросы	+	-	-
Поддержка разных СУБД	+	+	+
Независимость от веб-фреймворка	+	-	+
Развитая экосистема	+	+	-
Поддержка и ORM и SQL-конструктора	+	-	-
Автоматическая работа с миграциями	-	+	-

1.2.6. Выбор веб-фреймворка

Веб-фреймворк Django.

Django – это один из наиболее популярных веб-фреймворков для языка программирования Python, предназначенный для разработки полнофункциональных веб-приложений. Он предоставляет разработчику большое количество встроенных компонентов, включая систему маршрутизации, механизм работы с базами данных, систему аутентификации пользователей и административную панель.

К преимуществам Django можно отнести:

– наличие большого количества встроенных инструментов для разработки веб-приложений;

– встроенную ORM для работы с базами данных;

– развитую систему безопасности и аутентификации;

– большое сообщество разработчиков и подробную документацию.

Однако данный фреймворк имеет и ряд недостатков. Django является достаточно «тяжёлым» фреймворком, так как включает множество компонентов, которые могут быть избыточными для небольших проектов. Кроме того, структура проекта строго регламентирована, что иногда ограничивает гибкость разработки.

Веб-фреймворк Flask.

Flask – это лёгкий веб-фреймворк для языка Python, предназначенный для создания веб-приложений и API. В отличие от Django, Flask не содержит большого количества встроенных компонентов и предоставляет разработчику только базовый функционал для обработки HTTP-запросов.

Преимущества:

– простота и минималистичная структура;

– высокая гибкость при разработке приложений;

– возможность подключать только необходимые библиотеки и компоненты;

– удобство создания небольших веб-сервисов.

К недостаткам Flask можно отнести отсутствие многих встроенных инструментов, которые приходится подключать отдельно. Это может увеличивать время разработки крупных проектов и усложнять поддержку приложения.

Веб-фреймворк FastAPI.

FastAPI – современный веб-фреймворк для разработки API на языке Python, ориентированный на высокую производительность и удобство разработки. Он построен на основе современных технологий Python и широко применяется для создания REST-сервисов.

К основным преимуществам FastAPI относятся:

– высокая скорость работы и обработки запросов;
– автоматическая генерация документации API;
– поддержка асинхронного программирования;
– удобная работа с типами данных;
– валидация запросов;
– хорошая интеграция с современными библиотеками Python.

К недостаткам FastAPI можно отнести сравнительно меньший срок существования по сравнению с Django, а также меньший набор встроенных компонентов. Для реализации некоторых функций требуется подключение дополнительных библиотек.

После анализа возможностей различных веб-фреймворков для реализации серверной части приложения был выбран FastAPI. Данный фреймворк позволяет быстро создавать высокопроизводительные API-сервисы, обеспечивает удобную работу с типами данных и автоматически формирует документацию. Использование FastAPI хорошо подходит для разработки серверной части веб-приложения, взаимодействующего с клиентским интерфейсом и базой данных PostgreSQL [7]. Сравнение различных веб-фреймворков приведено в таблице 4.

Таблица 4 – плюсы и минусы разных веб-фреймворков

Характеристика	Django	Flask	FastAPI
Подходит для создания REST API	+-	+	+
Поддержка асинхронного программирования	-	-	+
Автоматическая генерация документации API	-	-	+
Большое количество встроенных компонентов	+	-	-
Встроенная ORM	+	-	-
Гибкость структуры проекта	-	+	+
Удобная валидация входных данных	-	-	+
Встроенная система аутентификации	+	-	-

1.2.7. Выбор очереди задач

Распределенная асинхронная очередь задач Taskiq.

Taskiq – это современная библиотека для выполнения отложенных и фоновых асинхронных задач в приложениях на языке Python. Такие задачи могут включать отправку уведомлений, обработку данных, формирование отчётов и выполнение различных операций, требующих дополнительного времени.

К основным преимуществам Taskiq можно отнести:

– поддержку асинхронного выполнения;

– поддержка интеграции с современными Python-фреймворками, включая FastAPI;

– поддержка большинства брокеров сообщений (RabbitMQ, NATS и др.);

– простую архитектуру и удобство использования;

– возможность масштабирования обработки задач.

К недостаткам Taskiq можно отнести сравнительно небольшую распространённость по сравнению с более зрелыми системами обработки задач. Также библиотека имеет меньшее количество обучающих материалов и примеров использования.

Распределенная очередь задач Celery.

Celery – одна из наиболее известных систем распределённого выполнения фоновых задач для Python. Она широко применяется в веб-приложениях для выполнения длительных операций вне основного потока выполнения программы.

Преимущества Celery:

– высокая стабильность и широкое распространение;

– поддержка различных брокеров сообщений (RabbitMQ, Redis и др.);

– возможность распределённого выполнения задач на нескольких серверах;

– развитая экосистема инструментов и библиотек.

Недостатками Celery являются достаточно сложная настройка и конфигурация системы. Кроме того, в классической реализации Celery ориентирован преимущественно на синхронную модель выполнения задач, что может усложнять использование в полностью асинхронных приложениях.

Распределенная очередь задач RQ (Redis Queue).

RQ – это библиотека для выполнения фоновых задач в Python, основанная на использовании Redis в качестве брокера сообщений. Она предназначена для организации очередей задач и их выполнения в отдельных рабочих процессах.

Преимущества RQ:

– простота установки и настройки;

– понятная архитектура работы с очередями задач;

– хорошая интеграция с Redis;

– удобство использования в небольших проектах.

К недостаткам RQ можно отнести ограниченную функциональность по сравнению с более сложными системами очередей задач. Кроме того, RQ в

основном ориентирован на синхронное выполнение задач и имеет меньше возможностей для масштабирования распределённых систем.

После анализа существующих решений для обработки фоновых и отложенных задач была выбрана библиотека Taskiq. Данное решение хорошо подходит для современных асинхронных веб-приложений и обеспечивает удобную интеграцию с фреймворком FastAPI [8].

Таблица 5 – плюсы и минусы разных очередей задач

Характеристика очереди	Taskiq	Celery	RQ
Используется для фоновых и отложенных задач	+	+	+
Используется для периодических задач	+	+	+
Поддержка асинхронного выполнения	+	-	+-
Хорошая интеграция с FastAPI	+	+-	+-
Поддержка разных брокеров сообщений	+	+	-
Простота настройки	+	-	+
Широкая распространенность	-	+	-

1.2.8. Выбор брокера сообщений

Брокер сообщений NATS.

NATS – это высокопроизводительная система обмена сообщениями, предназначенная для организации взаимодействия между различными компонентами распределённых приложений. Она используется для передачи сообщений между сервисами, обработки событий и организации очередей задач. Одной из особенностей NATS является наличие расширения JetStream, которое добавляет персистентность, повторную доставку и работу с потоками данных.

К основным преимуществам NATS можно отнести:

– высокую производительность и малую задержку передачи сообщений;

– поддержку масштабирования и распределённых систем;

– наличие встроенного механизма JetStream для хранения и обработки потоков сообщений;

– удобную интеграцию с современными библиотеками обработки фоновых задач, включая Taskiq.

К недостаткам NATS можно отнести сравнительно меньшую распространённость по сравнению с некоторыми другими брокерами сообщений.

Брокер сообщений RabbitMQ.

RabbitMQ – один из наиболее популярных брокеров сообщений, реализующий протокол AMQP и применяемый для построения распределённых систем и обработки очередей задач. Он обеспечивает надёжную доставку сообщений и поддерживает различные модели обмена данными между сервисами.

Преимущества RabbitMQ:

– высокая надёжность доставки сообщений;

– развитая система маршрутизации сообщений;

– поддержка очередей, обменников и различных шаблонов обмена данными;

– наличие механизмов подтверждения доставки сообщений.

К недостаткам RabbitMQ можно отнести более сложную настройку и администрирование. Кроме того, механизм долговременного хранения сообщений требует дополнительной конфигурации и может влиять на производительность системы.

Брокер сообщений Apache Kafka.

Apache Kafka – распределённая платформа потоковой обработки данных, предназначенная для обработки больших потоков сообщений в режиме реального времени. Она часто используется в системах с высокой нагрузкой и большими объёмами данных.

Преимущества Kafka:

– высокая производительность при работе с большими потоками данных;

– встроенные механизмы хранения сообщений и журналирования событий;

– высокая масштабируемость и отказоустойчивость;

– поддержка потоковой обработки данных.

Недостатками Kafka являются сложность настройки и администрирования, а также более высокие требования к инфраструктуре. Для небольших проектов использование Kafka может быть избыточным.

После анализа возможностей различных брокеров сообщений был выбран NATS с использованием технологии JetStream. Данное решение отличается высокой производительностью, простотой развертывания и наличием встроенного механизма хранения сообщений. Кроме того, NATS хорошо интегрируется с библиотекой Taskiq, используемой для обработки фоновых задач [9].

Таблица 6 – плюсы и минусы разных брокеров

Характеристика брокера	NATS	RabbitMQ	Apache Kafka
Подходит для обмена сообщениями между сервисами	+	+	+
Подходит для фоновых и отложенных задач	+	+	-
Поддержка высокой производительности и низкой задержки	+	+-	+
Простота установки и настройки	+	-	-
Подходит для небольших и средних проектов	+	+	+-
Поддержка потоковой обработки сообщений	+	-	+
Встроенное хранение сообщений	+	+	+
Наличие подтверждения доставки сообщений	+	+	+
Высокие требования к инфраструктуре	-	-	+

1.2.9. Фреймворк для внедрения зависимостей Dishka

Dishka – это библиотека реализующая механизм внедрения зависимостей. Данный подход используется при построении программной архитектуры и позволяет управлять зависимостями между компонентами системы. Основная идея внедрения зависимостей заключается в том, что объекты не создают необходимые им зависимости самостоятельно, а получают их извне через специальный контейнер.

Использование Dishka позволяет реализовать более чистую архитектуру приложения и уменьшить связанность между его компонентами. Это особенно важно при построении многослойных систем, где необходимо разделять различные уровни логики приложения.

К преимуществам Dishka можно отнести:

– удобное управление зависимостями между компонентами системы;
– поддержку контейнеров и областей жизни объектов;
– упрощение тестирования отдельных компонентов системы;
– снижение связности между слоями архитектуры.

К недостаткам Dishka можно отнести относительно небольшую распространённость по сравнению с более известными решениями. Кроме того, использование механизма внедрения зависимостей требует более тщательного проектирования архитектуры приложения [10].

1.3 Проектирование информационной системы

1.3.1 Архитектура приложения

Разрабатываемая система представляет собой клиент-серверное веб-приложение, включающее пользовательский интерфейс, серверную часть, базу данных, брокер сообщений и асинхронную очередь задач. Общая архитектура приложения построена таким образом, чтобы разделить пользовательскую логику, бизнес-логику, хранение данных и выполнение служебных процессов на независимые части, что упрощает сопровождение, развертывание и масштабирование системы [11].

В состав приложения входят следующие основные компоненты:

– клиентская часть;

– серверная часть;

– веб-сервер и обратный прокси Nginx;

– система управления базами данных PostgreSQL;

– брокер сообщений NATS;

– система фонового выполнения задач Taskiq.

Клиентская часть приложения реализует пользовательский интерфейс и предназначена для взаимодействия пользователя с системой через веб-браузер. На стороне клиента отображаются страницы сайта, формы авторизации и регистрации, личные кабинеты пользователей, расписание, курсы, анкеты и иные элементы интерфейса. Frontend-приложение разворачивается в отдельном Docker-контейнере.

Серверная часть приложения реализует обработку запросов от клиентской части, выполнение бизнес-логики, работу с пользователями, расписанием, курсами, нормативами, уведомлениями и другими сущностями системы. Backend также разворачивается в отдельном Docker-контейнере. Такое разделение на отдельные контейнеры позволяет независимо обновлять и сопровождать клиентскую и серверную части приложения.

В качестве точки входа в систему используется Nginx, который выполняет функции веб-сервера и обратного прокси. Nginx принимает входящие HTTP-запросы от пользователей и перенаправляет их к соответствующему компоненту системы. Запросы, связанные с отображением пользовательского интерфейса, направляются к frontend-приложению, а запросы к API – к backend-приложению.

Для хранения данных используется PostgreSQL, развернутая непосредственно на сервере. В базе данных хранятся сведения о пользователях, ролях, сотрудниках, клиентах, курсах, расписании, тренировочных слотах, результатах нормативов, статьях и уведомлениях. Размещение СУБД вне контейнера позволяет обеспечить постоянство хранимых данных, упростить резервное копирование и не связывать жизненный цикл базы данных с жизненным циклом контейнеров приложения.

NATS используется в приложении как брокер сообщений для хранения и передачи задач. В рамках данного проекта NATS применяется не для распределения нагрузки, а для организации служебных и отложенных операций внутри серверной части приложения.

Taskiq используется для выполнения фоновых, отложенных и периодических задач. Задачи передаются в NATS, где ожидают обработки, после чего выполняются рабочими процессами Taskiq. Такой механизм может использоваться, например, для отзыва пользовательских сессий.

Взаимодействие компонентов системы осуществляется следующим образом.

Пользователь открывает веб-приложение в браузере и отправляет запрос на сервер. Этот запрос сначала поступает в Nginx. Далее, в зависимости от типа запроса, Nginx перенаправляет его либо на frontend, либо на backend. Если запрос связан с получением или изменением данных, frontend отправляет HTTP-запрос к серверной части приложения. Backend обрабатывает запрос, при необходимости обращается к базе данных PostgreSQL, получает или изменяет данные и возвращает результат клиентской части. Если в ходе обработки требуется выполнить отложенную или фоновую операцию, backend формирует соответствующую задачу, отправляет ее в NATS, после чего Taskiq получит ее и выполняет в

отдельном рабочем процессе. Схема развертывания системы представлена на рисунке 7.

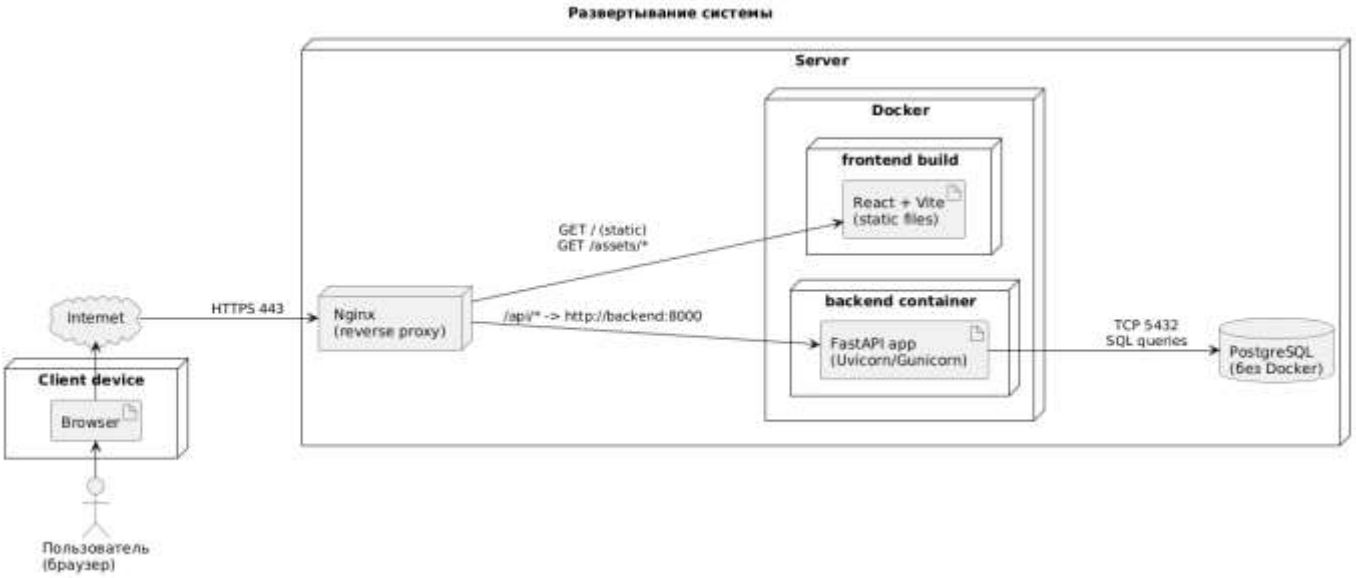


Рисунок 7 – Диаграмма развертывания приложения

1.3.2 Архитектура серверной части приложения

Серверная часть приложения разделена на несколько частей:

– слой представления (/api)

Слой представления реализован в виде REST API с использованием фреймворка FastAPI. Он отвечает за обработку HTTP-запросов, получение данных от клиентской части приложения и формирование ответов. Именно через этот слой frontend взаимодействует с backend.

Основными компонентами данного слоя являются:

– FastAPI-приложение – основа серверной части, через которую настраиваются маршруты, зависимости и общая работа API;

– роутеры – модули, в которых описываются конкретные URL и методы запросов, например авторизация, работа с курсами, расписанием, клиентами и тренерами;

– валидация входных данных – выполняется при получении запросов, чтобы в систему попадали только корректные данные;

– аутентификация и авторизация – используются для проверки, кто именно обращается к системе и какие действия ему разрешены;

– обработчики ответов и ошибок – позволяют возвращать клиенту понятный результат выполнения запроса.

Слой представления является внешней точкой входа в приложение. Он принимает запрос, проверяет его корректность, определяет пользователя и передает управление в следующий слой, где уже выполняется основная логика.

– прикладной слой (/application)

В прикладном слое сосредоточена основная бизнес-логика приложения. Именно здесь реализуются основные сценарии работы системы: регистрация и авторизация пользователей, создание курсов, работа с расписанием, запись клиентов на курс, фиксация посещаемости, работа с нормативами и уведомлениями.

Главной особенностью этого слоя является то, что он не зависит от конкретных технических средств. Здесь не описываются детали работы с FastAPI, PostgreSQL или другими библиотеками. В прикладном слое описывается именно логика предметной области – то есть что система должна делать.

Основные компоненты прикладного слоя:

– use cases – сценарии использования системы, которые описывают отдельные действия, например запись клиента на курс или создание тренировочного слота;

– абстрактные интерфейсы (ports) – контракты, через которые прикладной слой взаимодействует с внешними компонентами, например с репозиториями или сервисами;

– схемы данных на Pydantic – используются для обмена данными между слоями, а также для удобного и понятного описания структуры запросов и ответов;

Использование use cases позволяет разделить функциональность на понятные сценарии, а применение интерфейсов делает код менее связанным. Благодаря этому можно изменять техническую реализацию, не затрагивая основную логику приложения.

– инфраструктурный слой (/infrastructure)

Инфраструктурный слой отвечает за техническую реализацию всех внешних зависимостей приложения. Если в прикладном слое описано, что должно быть сделано, то в инфраструктурном слое реализуется, как именно это будет выполняться.

В данном слое находятся:

– реализации репозиториев – компоненты, которые работают с базой данных PostgreSQL через ORM SQLAlchemy и выполняют операции чтения, записи, изменения и удаления данных;

– реализации сервисов – классы, которые выполняют прикладные технические задачи, например работу с токенами, паролями, уведомлениями и другими внешними механизмами;

– DI-контейнер – механизм внедрения зависимостей, который связывает интерфейсы из прикладного слоя с их реальными реализациями;

– логирование – настройка журналирования событий и ошибок, необходимая для контроля работы системы;

– работа с Taskiq – настройка и использование фоновых, отложенных и периодических задач;

– взаимодействие с брокером сообщений – передача задач в NATS для их последующей обработки.

Таким образом, инфраструктурный слой обеспечивает связь прикладной логики с реальными инструментами: базой данных, системой фоновых задач, логированием и настройками приложения.

В результате вся архитектура backend-приложения разделяется на понятные уровни: слой представления принимает запросы, прикладной слой обрабатывает бизнес-логику, а инфраструктурный слой реализует работу с конкретными технологиями. Такое разделение делает проект более устойчивым, удобным для сопровождения и дальнейшего развития [12]. Архитектура приложения представлена на рисунке 8.

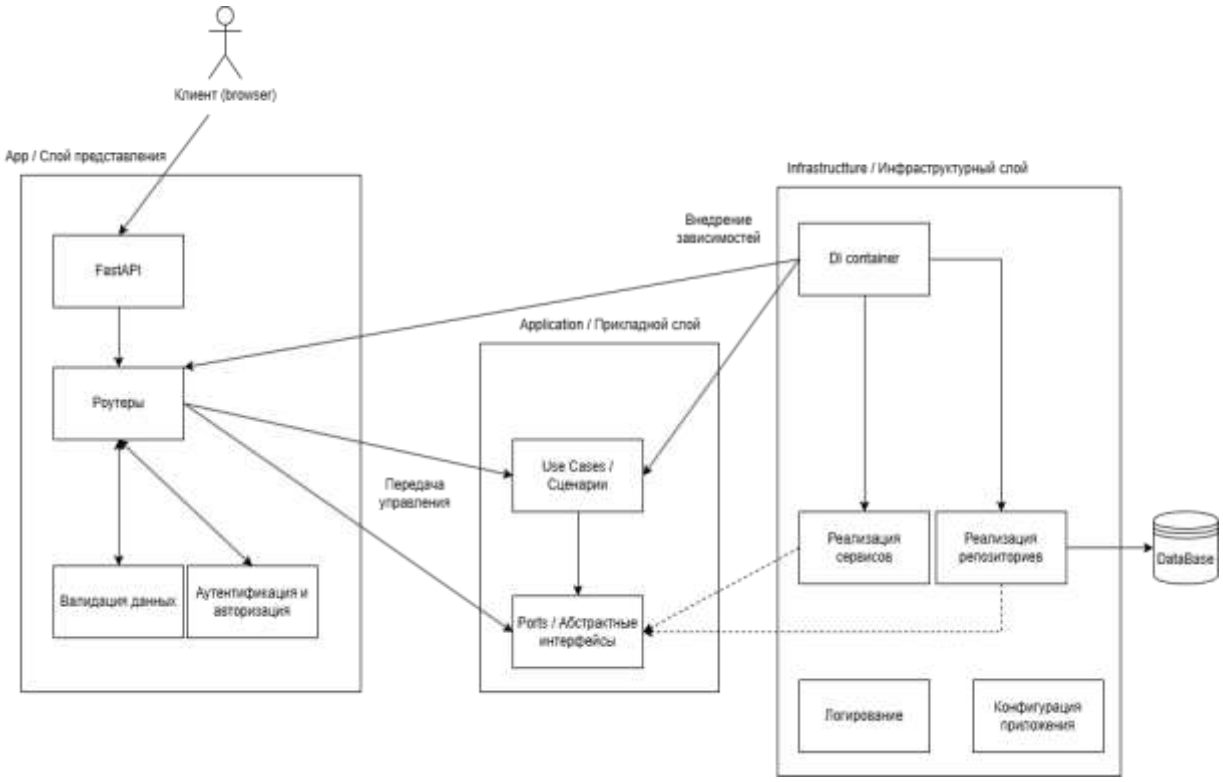


Рисунок 8 – Архитектура приложения

На рисунке 9 представлено взаимодействие компонентов приложения в виде диаграммы последовательности.

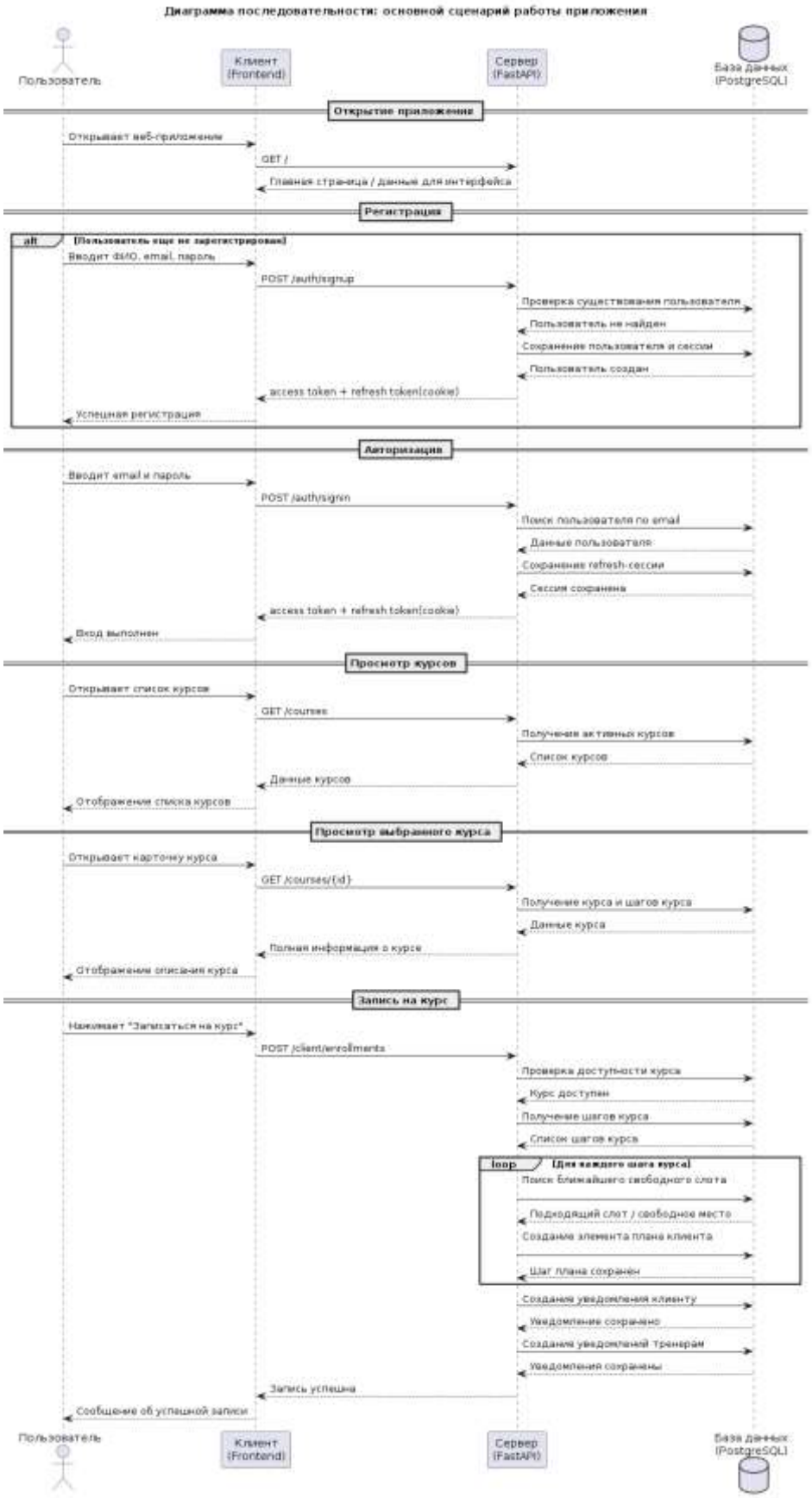


Рисунок 9– Диаграмма последовательности

Диаграмма классов, представляющая модель данных, изображена на рисунках 10-11.

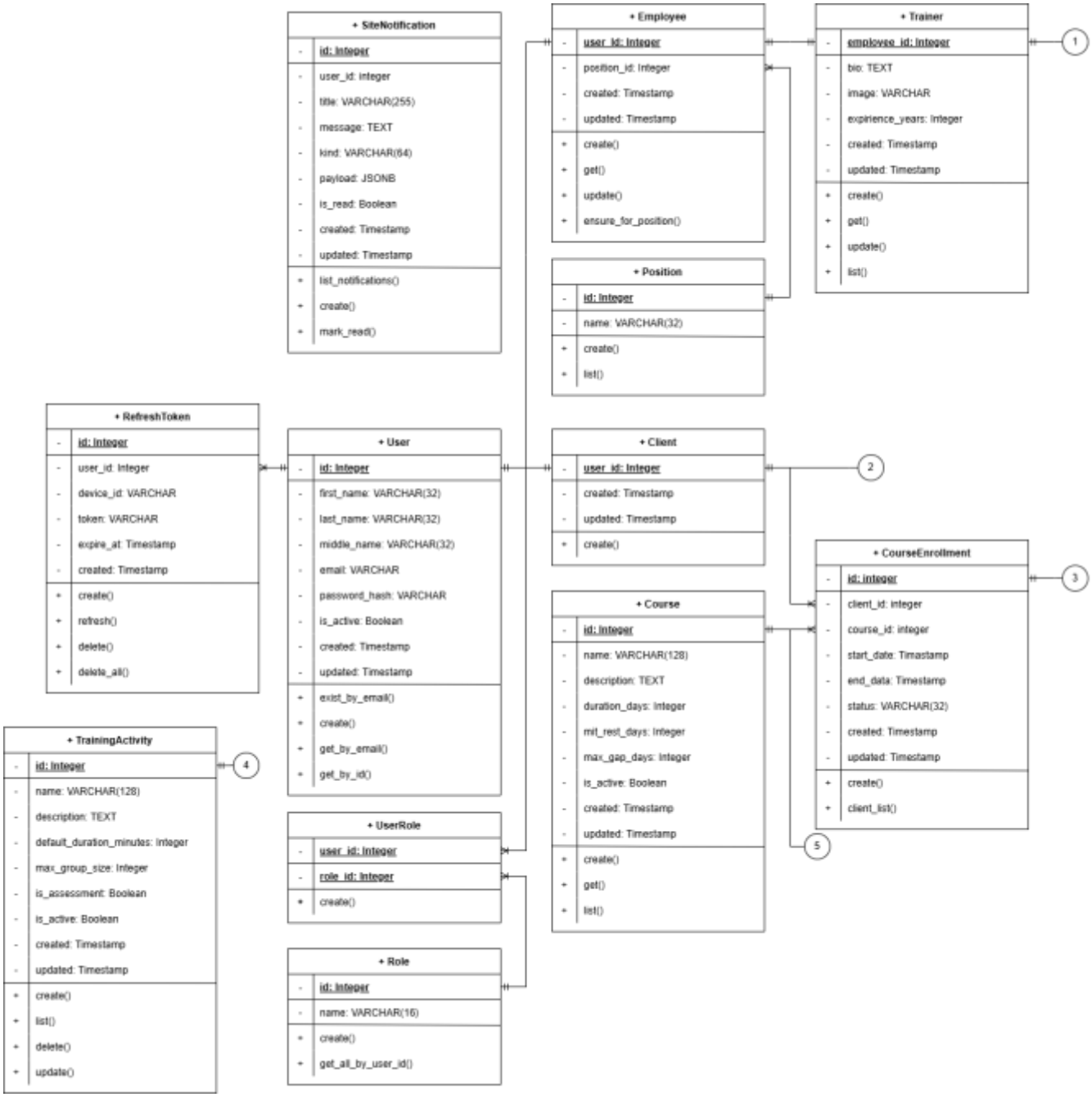


Рисунок 10 – Диаграмма классов (часть 1)

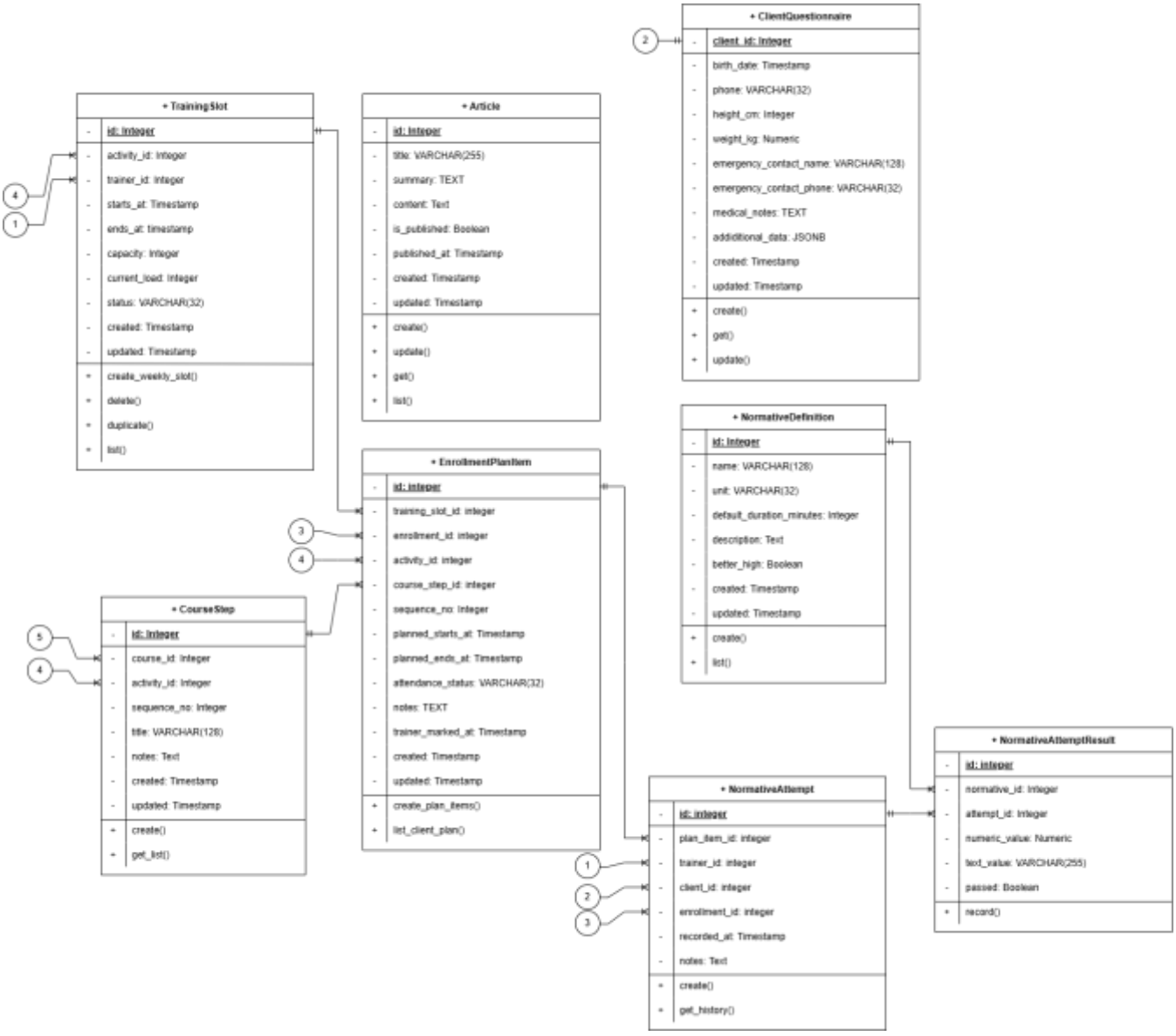


Рисунок 11 – Диаграмма классов (часть 2)

На рисунке 12 изображена диаграмма деятельности тренера.

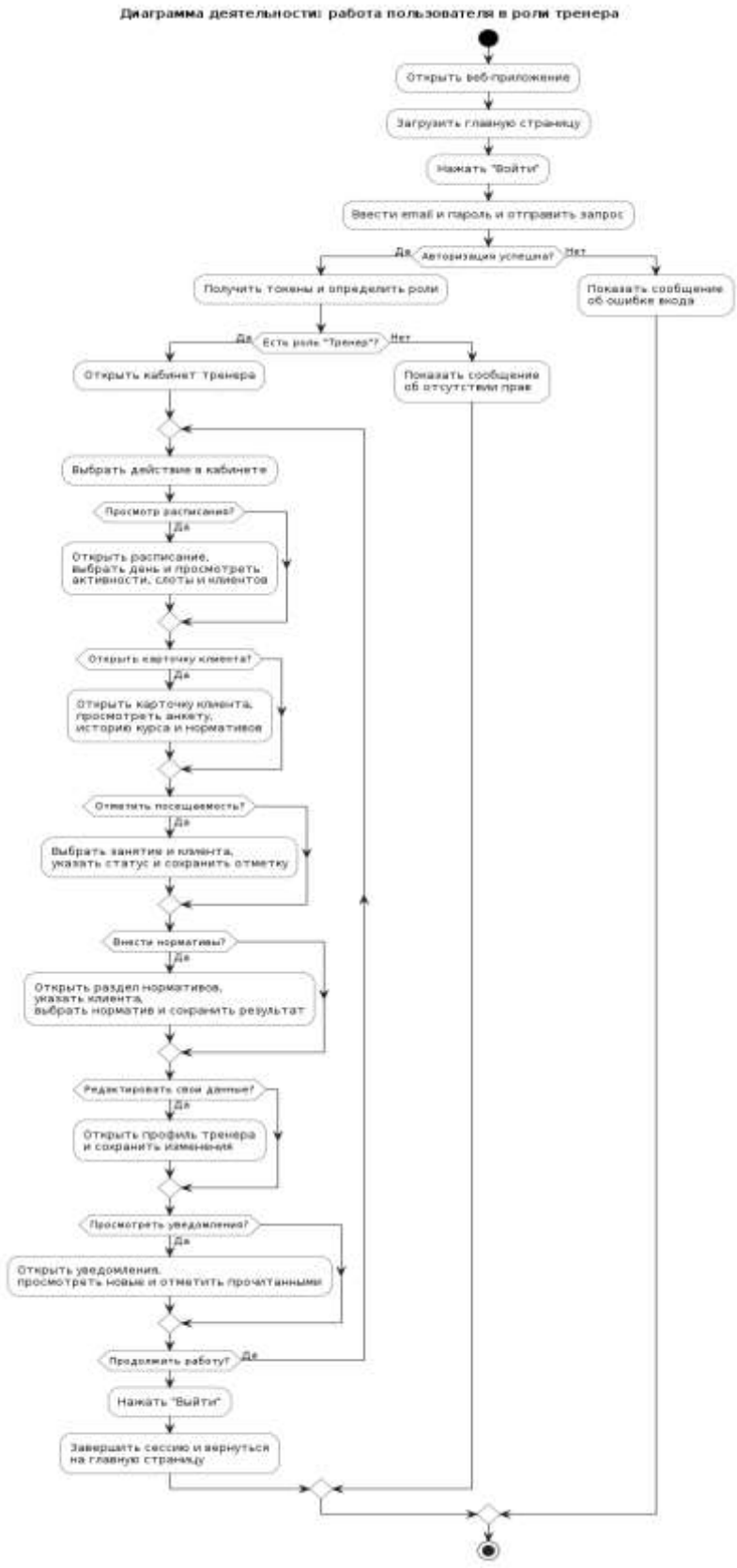


Рисунок 12 – Диаграмма деятельности

1.3.3 Проектирование базы данных

Для хранения информации необходимо реализовать базу данных со следующими таблицами:

– users – таблица, содержащая учетные данные пользователей системы (таблица 1);

– role – таблица, содержащая перечень ролей пользователей системы (таблица 2);

– user_role – связующая таблица между пользователями и ролями (таблица 3);

– refresh_token – таблица, содержащая активные пользовательские сессии (таблица 4);

– client – таблица, содержащая сведения о пользователях, зарегистрированных в системе в роли клиентов центра подготовки (таблица 5);

– client_questionnaire – таблица, хранящая анкетные данные клиента (таблица 6);

– position – таблица, содержащая перечень должностей сотрудников организации (таблица 7);

– employee – таблица, содержащая сведения о сотрудниках организации (таблица 8)

– trainer – таблица, содержащая сведения о тренерах (таблица 9);

– training_activity – таблица, содержащая информацию о видах тренировочной активности (таблица 10);

– normative_definition – таблица, содержащая справочник нормативов, которые могут фиксироваться в ходе подготовки клиента (таблица 11);

– course – таблица, содержащая сведения о курсах подготовки, доступных для прохождения клиентами (таблица 12);

– course_step – таблица, содержащая шаги курса подготовки, которые необходимо пройти клиенту в рамках выбранного курса (таблица 13);

– slot_template – таблица, содержащая шаблоны расписания занятий (таблица 14);

– training_slot – таблица, содержащая конкретные элементы расписания занятий (таблица 15);

– course_enrollment – таблица, содержащая записи клиентов на курсы подготовки (таблица 16);

– enrollment_plan_item – таблица, содержащая индивидуальный план клиента в рамках курса (таблица 17);

– normative_attempt – таблица, содержащая сведения о фактах сдачи нормативов клиентом в процессе подготовки (таблица 18);

– normative_attempt_result – таблица, содержащая результаты конкретных нормативов в рамках одной попытки сдачи (таблица 19);

– article – таблица, содержащая статьи и информационные материалы, публикуемые на главной странице системы (таблица 20);

– site_notification – таблица, содержащая уведомления пользователей внутри веб-приложения (таблица 21);

Таблица 7 – Атрибуты таблицы users

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор пользователя	PK	1
first_name	VARCHAR(64)	Имя пользователя		Иван
last_name	VARCHAR(64)	Фамилия пользователя		Петров
middle_name	VARCHAR(64)	Отчество пользователя		Иванович
email	VARCHAR(255)	Адрес электронной почты для входа		ivan@example.com
password_hash	VARCHAR	Хэш пароля пользователя		$2b$12$...

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
is_active	BOOLEAN	Признак активности учетной записи		true
created	TIMESTAMP WITH TIME ZONE	Дата и время создания		2026-04-17 12:30:00+03
updated	TIMESTAMP WITH TIME ZONE	Дата и время последнего изменения		2026-04-17 13:00:00+03

Таблица 8 – Атрибуты таблицы role

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор роли	PK	1
name	VARCHAR(32)	Название		admin

Таблица 9 – Атрибуты таблицы user_role

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
user_id	INTEGER	Идентификатор пользователя	PK	1
role_id	INTEGER	Идентификатор роли	PK	2

Таблица 10 – Атрибуты таблицы refresh_token

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор записи сессии	PK	1
user_id	INTEGER	Идентификатор пользователя	FK	1
device_id	VARCHAR	Уникальный идентификатор устройства		browser-7f31a
token	UUID	токен сессии		018f1d5f-...

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
expire_at	TIMESTAMP	Дата и время истечения сессии		2026-05-17 12:30:00+03
created	TIMESTAMP	Дата и время создания сессии		2026-04-17 12:30:00+03

Таблица 11 – Атрибуты таблицы client

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор клиента	PK	5
created	TIMESTAMP	Дата и время создания записи клиента		2026-04-17 12:40:00+03
updated	TIMESTAMP	Дата и время последнего изменения записи клиента		2026-04-17 13:10:00+03

Таблица 12 – Атрибуты таблицы client_questionnaire

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
client_id	INTEGER	Идентификатор клиента	PK	1
birth_date	TIMESTAMP	Дата рождения		2000-01-01
phone	VARCHAR(32)	Номер телефона		+79991232112
height_cm	INTEGER	Рост		155
weight_kg	NUMERIC	Вес		90
Emergency_contact_name	VARCHAR(128)	Контактное лицо для экстренной связи		Петровая Мария

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
emergency_contact_phone	VARCHAR(32)	Телефон экстренного контакта		+79991232112
medical_notes	TEXT	Медицинские ограничения и примечания		Ограничение нагрузки на колено
additional_data	JSONB	Дополнительные анкетные данные		{"sport_level":"beginner"}
created	TIMESTAMP WITH TIME ZONE	Дата и время создания		2026-04-17 12:45:00+03
updated	TIMESTAMP WITH TIME ZONE	Дата и время изменения		2026-04-17 13:15:00+03

Таблица 13 – Атрибуты таблицы position

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор должности	PK	1
name	VARCHAR(64)	Наименование должности		trainer
description	TEXT	Описание должности		Тренер по физической подготовке
is_active	BOOLEAN	Признак активности должности		true
created	TIMESTAMP WITH TIME ZONE	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP WITH TIME ZONE	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 14 – Атрибуты таблицы employee

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор сотрудника	PK	2
position_id	INTEGER	Идентификатор должности	FK	1
hired_at	TIMESTAMP	Дата приема на работу		2000-01-01
fired_at	TIMESTAMP	Дата увольнения		2000-01-01
is_active	BOOLEAN	Признак активности сотрудника		True
created	TIMESTAMP	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 15 – Атрибуты таблицы trainer

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор тренера	PK	1
bio	TEXT	Информация о тренере		Специалист по атлетике
image	VARCHAR	Путь до изображения		/123.jpg
experience_years	INTEGER	Стаж работы		6
created	TIMESTAMP	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 16 – Атрибуты таблицы training_activity

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор вида активности	PK	1
name	VARCHAR(128)	Название		Легкая атлетика
desctiption	TEXT	Описание		Беговые управжнения
duration_minutes	INTEGER	Длительность в минутах		60
max_group_size	INTEGER	Максимальный размер группы		5
Ii_assessment	BOOLEAN	Является ли активность аттестационной		False
is_active	BOOLEAN	Признак активности записи		True
created	TIMESTAMP	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 17 – Атрибуты таблицы normative_definition

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор норматива	PK	1
name	VARCHAR(32)	Название		Бег 100м
unit	VARCHAR(32)	Единица измерения		Сек
value_type	VARCHAR(32)	Тип значения норматива		Number
description	TEXT	Описание		Время прохождения 100м

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
better_high	BOOLEAN	Является ли большее значение лучшим		False
created	TIMESTAMP	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 18 – Атрибуты таблицы course

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор курса подготовки	PK	1
name	VARCHAR(128)	Название		Спец. Подготовка
desctiption	TEXT	Описание		Курс базовой физической подготовки
duration_days	INTEGER	Длительность курса		30
is_active	BOOLEAN	Признак активности курса		True
created	TIMESTAMP	Дата и время создания		2026-04-17 12:00:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 12:30:00+03

Таблица 19 – Атрибуты таблицы course_step

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор шага курса	PK	1
course_id	INTEGER	Идентификатор курса	FK	1

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
activity_id	INTEGER	Идентификатор вида активности	FK	1
sequence_no	INTEGER	Порядковый номер шага в курсе		1
title	VARCHAR(128)	Название шага курса		Сдача нормативов
notes	TEXT	Примечание к шагу курса		Контрольное занятие
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00+03

Таблица 20 – Атрибуты таблицы slot_template

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор шаблона слота	PK	1
activity_id	INTEGER	Идентификатор активности	FK	1
trainer_id	INTEGER	Идентификатор тренера	FK	1
weekday	INTEGER	День недели, на который назначен шаблон		3
start_time	TIME	Время начала занятия		08:00:00
duration_minutes	INTEGER	Продолжительность		60
capacity	INTEGER	Вместимость группы		10
valid_from	DATE	Дата начала действия шаблона		2000-01-01
valid_to	DATE	Дата окончания действия шаблона		2000-01-01
is_active	BOOLEAN	Признак активности шаблона		True

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00

Таблица 21 – Атрибуты таблицы training_slot

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор конкретного слота расписания	PK	1
slot_template_id	INTEGER	Идентификатор шаблона	FK	1
activity_id	INTEGER	Идентификатор активности	FK	1
trainer_id	INTEGER	Идентификатор тренеера	FK	1
starts_at	TIMESTAMP	Дата и время начала занятия		2000-01-01 08:00:00
ends_at	TIMESTAMP	Дата и время окончания занятия		2000-01-01 08:00:00
capacity	INTEGER	Максимальное количество клиентов в слоте		5
current_load	INTEGER	Текущая загрузка слота		3
status	VARCHAR(32)	Статус слота		Open
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00+03

Таблица 22 – Атрибуты таблицы course_enrollment

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор Записи клиента на курс	PK	1
client_id	INTEGER	Идентификатор клиента	FK	1
course_id	INTEGER	Идентификатор курса	FK	1
start_date	DATE	Дата начала		2000-01-01
end_data	DATE	Дата окончания		2000-01-01
status	VARCHAR(32)	Статус прохождения курса		Active
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00+03

Таблица 23 – Атрибуты таблицы enrollment_plan_item

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор элемента индивидуального плана	PK	1
enrollment_id	INTEGER	Идентификатор записи на курс	FK	1
course_step_id	INTEGER	Идентификатор шага курса	FK	1
training_slot_id	INTEGER	Идентификатор слота расписания	FK	1
activity_id	INTEGER	Идентификатор активности	FK	1
sequence_no	INTEGER	Порядковый номер элемента плана		1
planned_starts_at	TIMESTAMP	Дата и время начала занятия		2026-04-17 10:35:00

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
planned_ends_at	TIMESTAMP	Дата и время окончания назначенного занятия		2026-04-17 10:35:00
attendance_status	VARCHAR(32)	Статус посещения		scheduled
notes	TEXT	Примечания к занятию		
trainer_marked_at	TIMESTAMP	Дата и время отметки тренером		2026-04-17 10:35:00+03
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00+03

Таблица 24 – Атрибуты таблицы normative_attempt

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор попытки сдачи норматива	PK	1
plan_item_id	INTEGER	Идентификатор занятия в плане клиента	FK	2
enrollment_id	INTEGER	Идентификатор записи на курс	FK	1
client_id	INTEGER	Идентификатор клиента	FK	1
trainer_id	INTEGER	Идентификатор тренера	FK	1
recorded_at	TIMESTAMP	Дата и время внесения результатов		2000-01-01 08:00:00
notes	TEXT	Примечание к попытке сдачи		Контрольный замер

Таблица 25 – Атрибуты таблицы normative_attempt

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор результата норматива	PK	1
attempt_id	INTEGER	Идентификатор попытки сдачи норматива	FK	1
normative_id	INTEGER	Идентификатор норматива	FK	1
numeric_value	NUMERIC	Числовое значение результата		100.12
text_value	VARCHAR(255)	Текстовое значение результата		
passed	BOOLEAN	Признак успешности выполнения норматива		true

Таблица 26 – Атрибуты таблицы normative_attempt

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор статьи	PK	1
title	VARCHAR(255)	Заголовок статьи		Правила питания
summary	TEXT	Краткое описание статьи		Основные рекомендации по питанию
content	TEXT	Основной текст статьи		Полный текст статьи
is_published	BOOLEAN	Дата и время публикации		true
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03
updated	TIMESTAMP	Дата и время изменения		2026-04-17 10:45:00+03

Таблица 27 – Атрибуты таблицы site_notification

Имя столбца	Тип данных	Описание	Роль атрибута	Пример
id	INTEGER	Идентификатор уведомления	PK	1
user_id	INTEGER	Идентификатор пользователя	FK	1
title	VARCHAR(255)	Заголовок		Успешная запись на курс
message	TEXT	Текст		Ваша запись подтверждена
kind	VARCHAR(64)	Тип уведомления		Enrollment_created
payload	JSONB	Дополнительные данные		{"course_id":1,"enrollment_id":3}
is_read	BOOLEAN	Признак прочтения уведомления		true
read_at	TIMESTAMP	Дата и время прочтения уведомления		2026-04-17 10:35:00+03
created	TIMESTAMP	Дата и время создания		2026-04-17 10:35:00+03

Физическая модель базы данных приведена на рисунках 13-14.

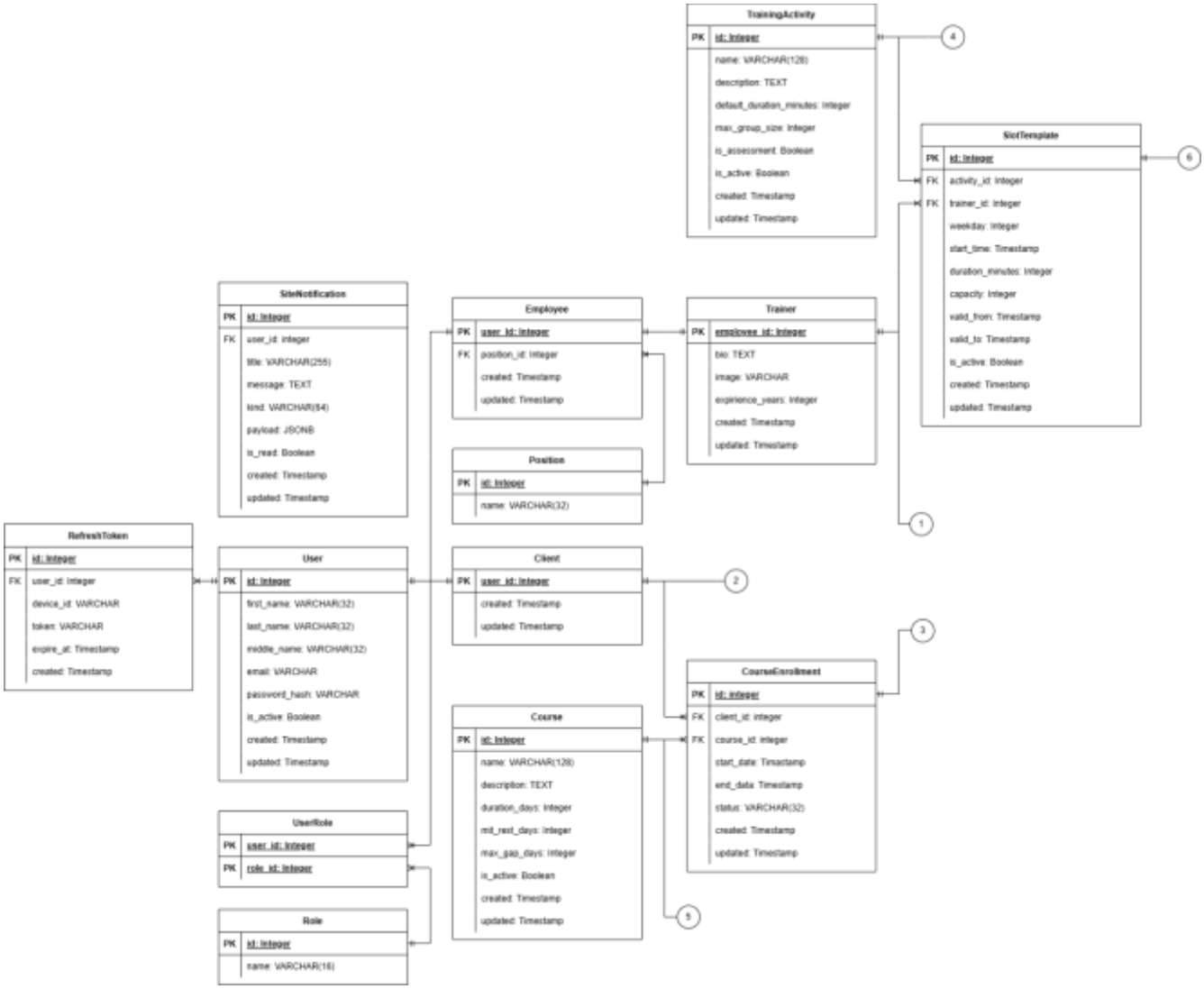


Рисунок 13 – Физическая модель БД (1 часть)

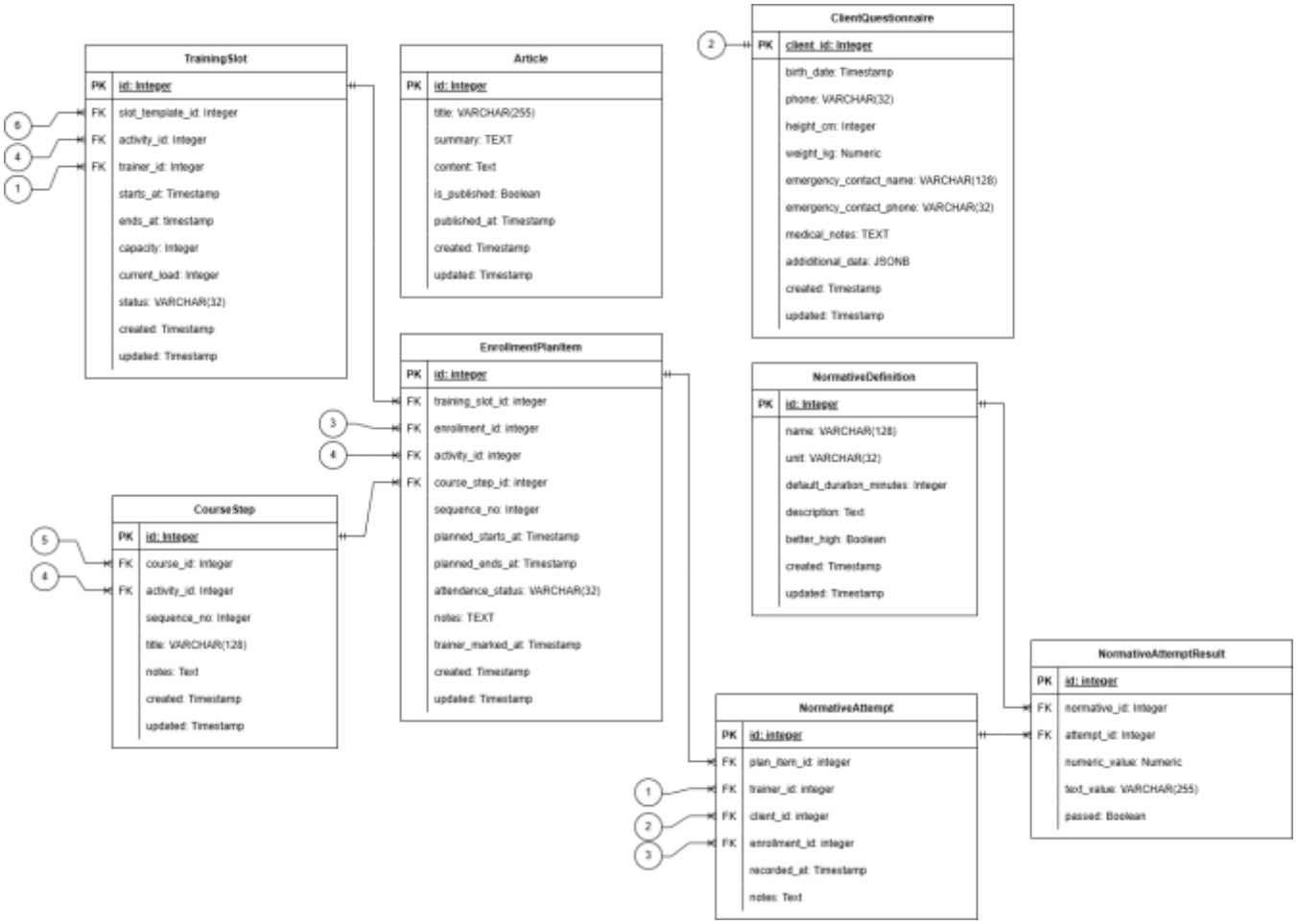


Рисунок 14 – Физическая модель БД (2 часть)

На рисунках 15-16 изображены логическая схема базы данных.

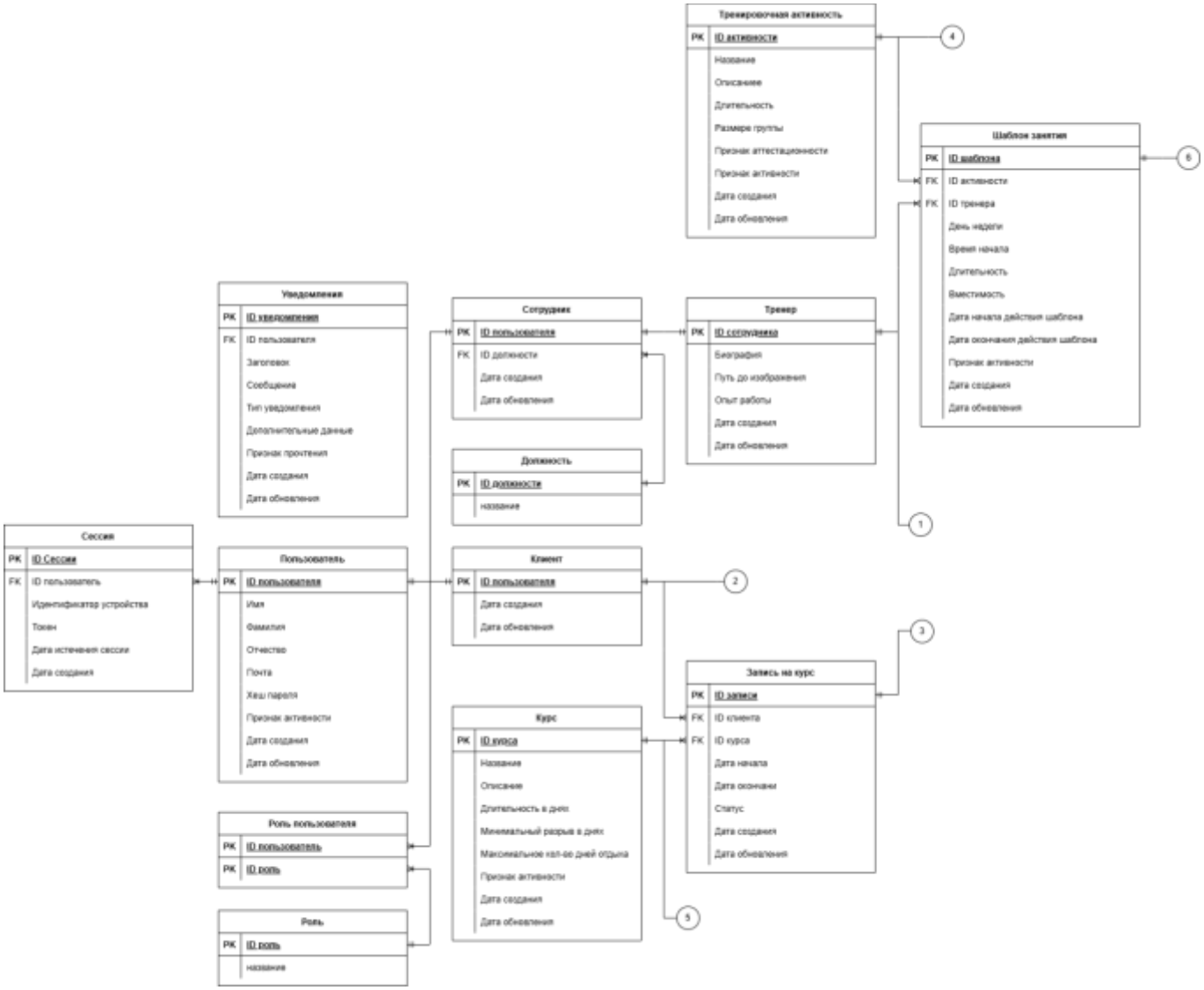


Рисунок 15 – Логическая схема БД (1 часть)

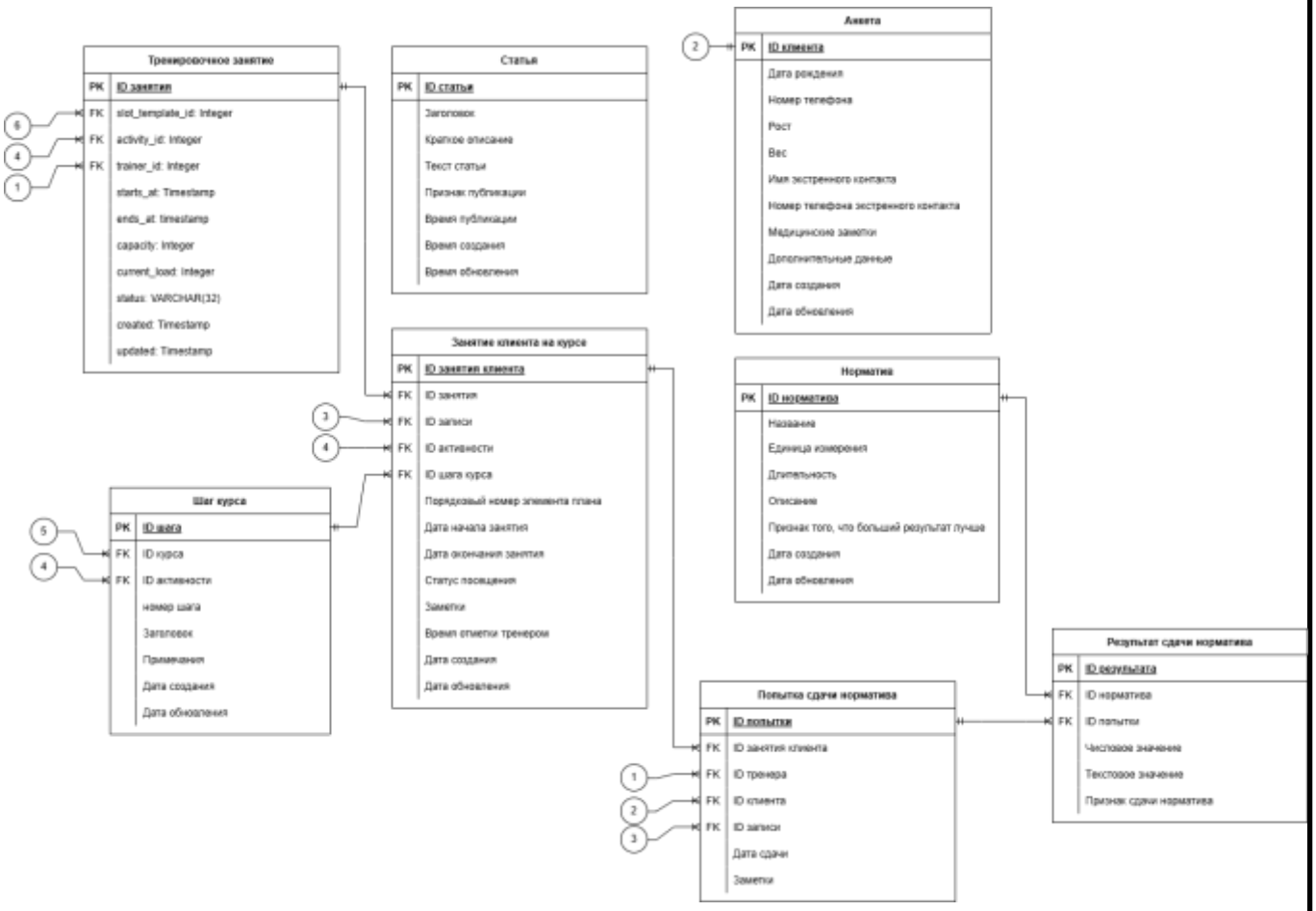


Рисунок 16 – Логическая схема БД (2 часть)

1.4 Разработка приложения

1.4.1 Реализация серверной части приложения

1.4.1.1 Настройка окружения

Для разработки и развертывания серверной части приложения было настроено программное окружение, включающее необходимые библиотеки и параметры конфигурации. В качестве языка программирования используется Python, а управление зависимостями осуществляется с помощью файла зависимостей requirements.txt, в котором зафиксированы версии используемых библиотек для обеспечения стабильности и воспроизводимости окружения.

Основой серверной части является фреймворк FastAPI, предназначенный для создания высокопроизводительных веб-приложений и API. Для запуска приложения и обработки входящих HTTP-запросов используется ASGI-сервер Uvicorn. Взаимодействие с базой данных PostgreSQL реализовано с использованием библиотеки SQLAlchemy, обеспечивающей объектно-реляционное отображение данных, а также асинхронного драйвера asyncpg, который используется для неблокирующего доступа к базе данных. Управление схемой базы данных и миграциями выполняется с помощью инструмента Alembic.

Для валидации и сериализации входных и выходных данных применяется библиотека Pydantic, которая также применяется для описания DTO-схем и передачи структурированных данных между слоями приложения. Аутентификация и работа с JSON Web Token реализованы с использованием библиотеки PyJWT. Для организации внедрения зависимостей и управления жизненным циклом компонентов приложения используется библиотека Dishka.

Список зависимостей в файле requirements.txt:

– alembic==1.17.2
– asyncpg==0.31.0
– dishka==1.7.2
– fastapi==0.126.0
– PyJWT==2.10.1

– python-dotenv==1.2.1
– SQLAlchemy==2.0.45
– structlog==25.5.0
– uuid6==2025.0.1
– uvicorn==0.40.0
– taskiq==0.12.4
– taskiq-nats==0.6.0

Конфигурация приложения осуществляется с использованием переменных окружения в файле .env, что позволяет отделить параметры окружения от исходного кода и упростить процесс развертывания. К основным переменным окружения относятся параметры подключения к базе данных (адрес, порт, имя базы данных, пользователь и пароль), секретные ключи для генерации и проверки JWT-токенов, а также настройки времени жизни токенов и параметры запуска серверного приложения.

Перед установки зависимостей сначала необходимо создать виртуальное окружение командой:

```
python -m venv venv
```

А затем активировать его:

```
venv\Scripts\activate
```

После активации следует установить все зависимости из файла requirements.txt:

```
pip install -r requirements.txt
```

1.4.1.2 Подготовка структуры проекта

Перед началом работы следует подготовить структуру проекта. Серверная часть проекта состоит из трех частей:

– слой представления (/api)

Реализован в виде REST API с использованием фреймворка FastAPI и отвечает за обработку HTTP-запросов, валидацию входных данных и формирование ответов. В данном слое определены маршруты (роутеры), которые принимают запросы от клиентской части приложения и передают управление в

соответствующие сценарии использования. Также на этом уровне выполняется проверка аутентификации и авторизации пользователей с использованием зависимостей FastAPI.

– прикладной слой (/application)

Бизнес-логика приложения сосредоточена в прикладном слое, который реализует основные сценарии работы системы, включая регистрацию и авторизацию пользователей, управление расписанием тренировок, создание, изменение и удаление тренировочных слотов, а также запись клиентов на занятия. Взаимодействие с внешними компонентами осуществляется через абстрактные интерфейсы (ports) и pydantic схемы, что обеспечивает слабую связность между слоями

– инфраструктурный слой (/infrastructure)

Инфраструктурный слой отвечает за реализацию технических деталей работы приложения и взаимодействие с внешними ресурсами. В данном слое размещены конкретные реализации репозиториев и сервисов, доступ к базе данных PostgreSQL, а также компоненты логирования и конфигурацией приложения. Данный слой реализует интерфейсы, определённые в прикладном слое, и подключается к нему посредством механизма внедрения зависимостей. Итоговая структура серверной части приложения представлена на рисунке 17.

Рисунок 17 – Структура серверной части проекта

1.4.1.3 Описание моделей базы данных и миграции

Для описания структуры базы данных в проекте используется библиотека SQLAlchemy, предоставляющая средства объектно-реляционного отображения и позволяющая описывать таблицы базы данных в виде Python-классов. Каждая модель соответствует отдельной таблице и содержит описание полей, их типов данных, первичных и внешних ключей, а также ограничений целостности. Такой подход упрощает работу с данными, повышает читаемость кода и обеспечивает согласованность между логической моделью и физической структурой базы данных.

Каждая модель наследуется от базового класса Base и имеет атрибут __tablename__, через который указывается название таблицы. Поля базы данных указываются через атрибуты класса с присвоением им значения функции mapped_column, типизированной через Mapped[type]. Например, поле id объявлено как Mapped[int] – целочисленный первичный ключ с автоинкрементом. В параметры функции mapped_column следует передать тип поля (Integer, String и др.), а также необязательные аргументы, например: nullable, primary_key, unique. Связи между таблицами задаются через атрибуты класса с использованием функции relationship(). Например, атрибут users описывает связь «многие ко многим» между ролями и пользователями: параметр secondary=”user_role” указывает промежуточную таблицу, через которую осуществляется связь, back_populates=”roles” задает обратную связь на стороне модели User, a uselist=True означает, что атрибут возвращает список объектов, а не единственный экземпляр.

Пример таблицы Role:

```
class Role(Base):
    __tablename__ = "role"
    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    name: Mapped[str] = mapped_column(String(32), nullable=False,
unique=True)
    users: Mapped[list["User"]] = relationship(
        secondary="user_role",
        back_populates="roles",
        uselist=True,
    )
```

Для управления изменениями структуры базы данных применяется инструмент Alembic, который используется для создания и применения миграций. Миграции позволяют версионировать схему базы данных и безопасно вносить изменения в структуру таблиц без потери данных. В рамках проекта Alembic используется для добавления новых таблиц, изменения существующих полей и управления связями между сущностями. Применение миграций обеспечивает воспроизводимость окружения и упрощает процесс развертывания и обновления серверной части приложения [13].

После описания всех моделей через ORM следует сделать миграции с помощью alembic:

Сначала alembic добавляется в проект и инициализируется, после чего в корневой директории создается служебная структура миграций:

```
alembic init migrations
```

В результате будут созданы файл alembic.ini и каталог migrations/, в котором хранятся миграции и конфигурационный файл env.py. Далее в env.py настраивается подключение к базе данных и указывается переменная target_metadata, которая позволяет alembic автоматически сравнивать текущее состояние моделей ORM со схемой базы данных:

```
config.set_main_option("sqlalchemy.url", settings.DATABASE_URL + "?async_fallback=True")
target_metadata = Base.metadata
```

Далее следует создать первую миграцию, которая включает все таблицы и связи, описанные в моделях:

```
alembic revision –autogenerate -m “initial”
```

В результате будут созданы файл \migrations\versions\<revision>_initial.py, в котором будут сформированы операции создания таблиц, внешних ключей и индексов.

Чтобы применить миграцию и создать таблицы в базе данных, выполняется команда:

```
alembic upgrade head
```

Результат миграции представлен на рисунке 18.

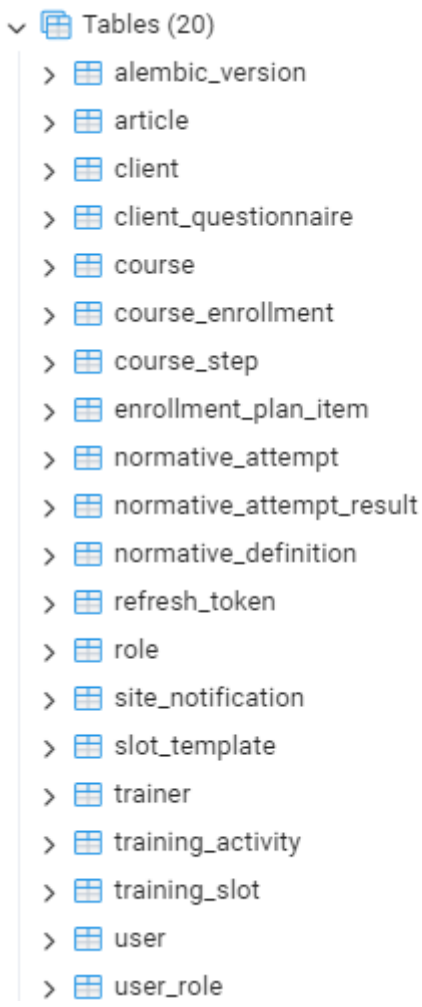


Рисунок 18 – Результат миграции

1.4.1.4 Реализация RESTful API

Протокол HTTP (HyperText Transfer Protocol) является прикладным протоколом передачи данных, используемым для обмена информацией между клиентом и сервером в распределённых информационных системах. В рамках веб-приложений HTTP применяется для передачи запросов от клиентской части к серверной и получения ответов с результатами обработки этих запросов. Протокол HTTP основан на модели «запрос–ответ» и поддерживает различные методы, такие как GET, POST, PUT, PATCH и DELETE, которые используются для выполнения операций над ресурсами системы.

REST API (Representational State Transfer Application Programming Interface) представляет собой архитектурный стиль построения программных интерфейсов,

использующий возможности протокола HTTP для взаимодействия между компонентами системы. В REST API доступ к данным и функциональности осуществляется через ресурсы, идентифицируемые с помощью URL, а операции над ними выполняются с использованием стандартных HTTP-методов. Данный подход обеспечивает простоту интеграции, масштабируемость и независимость клиентской и серверной частей приложения, что делает REST API удобным решением для построения современных веб-систем.

Реализация API находится в слое представления, каталог /api. Роутеры, отвечающие за прием запросов, валидацию данных, выполнение проверок безопасности и передачу управления в прикладной слой, находятся в каталоге /api/v1/endpoints.

Для группировки связанных эндпоинтов используется объект APIRouter, которому задается общий префикс маршрутов (например /auth) и тег, применяемый для документирования API.

Для примера рассмотрим роутер, отвечающий за авторизацию пользователя (auth_router) и обработчик, отвечающий за регистрацию пользователя:

```
auth_router = APIRouter(
    prefix="/auth",
    tags=["auth"],
    route_class=DishkaRoute,
)
@auth_router.post("/signup", response_model=AuthResponse)
async def registration(
    response: Response,
    user_data: SignUpRequest,
    client: RequestClient,
    signup_uc: FromDishka[SignUpUseCase]
):
    try:
        data = await signup_uc(
            registration_data=user_data,
            client_info=client
        )
        response.set_cookie(value=data.refresh_token,
**settings.COOKIE_SETTINGS)
        return AuthResponse(
            access_token=data.access_token,
            token_type=data.token_type
        )
    except UserAlreadyExist as uae:
```

```
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail=f"User already exist"
        )
```

Роутер auth_router создается как экземпляр класса APIRouter с префиксом /auth, тегом auth для документации и классом маршрута DishkaRoute, обеспечивающим интеграцию с контейнером зависимостей Dishka.

Далее описывается обработчик регистрации в виде асинхронной функции registration, который зарегистрирован через декоратор @auth_router.post(“/signup”, response_model=AuthResponse). Декоратор указывает, что обработчик принимает HTTP-запросы методом POST по маршруту /auth/signup, а параметр response_model задает схему ответа – FastAPI автоматически сериализует возвращаемый объект в соответствии с этой схемой и отобразит ее в автоматической документации.

Обработчик принимает следующие параметры: объект Response для управления HTTP-ответом, user_data типа SignUpRequest – тело запроса, client типа RequestClient – сведения о клиенте, извлеченные из запроса, а также signup_uc – экземпляр сценария использования SignUpUseCase, внедряемый с помощью контейнера зависимостей Dishka через аннотацию FromDishka.

При успешном выполнении сценария в HTTP-ответ установится cookie с refresh-токеном, а клиенту возвращается объект AuthResponse, содержащий access-токен и тип токена. В случае если пользователь с такими данными уже существует, перехватывается исключение UserAlreadyExist и клиенту возвращается HTTP-ответ со статусом 409 Conflict с соотвествующим сообщением об ошибке.

Все схемы, например AuthResponse и SingUpRequest, находятся в прикладном слое в каталоге /application/schemas и представляют собой классы, наследующиеся от BaseModel библиотеки Pydantic.

Рассмотрим схему SignUpRequest:

```
class SignUpRequest(BaseModel):
    email: EmailStr
    password: str
    first_name: str
    last_name: str
```

```
    middle_name: str | None
```

– SignUpRequest описывает тело входящего запроса и содержит следующие поля:

– email – тип EmailStr, который Pydantic автоматически проверяет на соответствие формату email;

– password, first_name, last_name – обязательные строковые поля;

– middle_name – необязательное строковое поле.

При получении запроса FastAPI автоматически десериализует тело запроса в экземпляр SignUpRequest и выполнит валидацию всех полей. В случае несоответствия данных клиенту возвращается ответ со статусом 422 Unprocessable Entity.

Для ограничения доступа к эндпоинтам на основе ролей пользователя в файле /api/security/role_filter.py реализован класс RoleRequired. Класс принимает в конструктор список допустимых ролей и реализован как вызываемый объект через метод __call__, что позволяет использовать его экземпляры напрямую в качестве зависимости FasttAPI через Depends.

Класс RoleRequired:

```
class RoleRequired:
    def __init__(self, allowed_roles: list[str]):
        self.allowed_roles = allowed_roles

    def __call__(self, user: UserBaseSchema =
Depends(get_token_payload)) -> UserBaseSchema:
        user_roles = set(user.roles)
        if not user_roles.intersection(self.allowed_roles):
            raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="Permission denied")
        return user
```

При каждом запросе метод __call__ получает данные текущего пользователя из JWT-токена через зависимость get_token_payload, после чего проверяет пересечение ролей пользователя с допустимыми ролями.

На основе RoleRequired создаются экземпляры класса для каждой роли: get_current_client, get_current_admin, get_current_trainer.

```
get_current_client = RoleRequired(["client", "admin"])
get_current_trainer = RoleRequired(["trainer", "admin"])
get_current_admin = RoleRequired(["admin"])
```

Чтобы ограничить доступ к эндпоинту нужно указать в функции-обработчике запроса параметр равный Depends(get_current_client).

Пример ограничения доступа к эндпоинту:

```
@admin_router.post("/positions", response_model=PositionRead)
async def create_position(
    payload: PositionCreate,
    usecase: FromDishka[CreatePositionUseCase],
    _: UserBaseSchema = Depends(get_current_admin),
):
    return await usecase(payload)
```

Сценарии использования, например SignUpUseCase, находятся в прикладном слое в каталоге /application/use_cases и реализуют конкретные бизнес-сценарии приложения. Их назначение – координировать вызовы сервисов прикладного слоя в нужном порядке и обеспечивать атомарное выполнение всех операций – если на любом шаге возникает ошибка, все изменения откатываются.

Для примера рассмотрим сценарий SignUpUseCase, который должен:

– Проверить существует ли пользователь с такой же почтой. Если существует, то вернуть ошибку.

– Создать пользователя.

– Создать клиента.

– Дать новому пользователю роль “client”

– Создать токен обновления, токен доступа, а также сессию пользователя.

– Вернуть схему SuccessSignUpSchema.

```
class SignUpUseCase:
    def __init__(
        self,
        user_service: UserServicePort,
        user_session_service: UserSessionServicePort,
        client_service: ClientServicePort,
        role_service: RoleServicePort,
        jwt_processor: JWTProcessorPort,
        settings: Settings,
        session: AsyncSession
    ):
        self.user_service = user_service
        self.user_session_service = user_session_service
        self.client_service = client_service
        self.role_service = role_service
```

```
        self.jwt_processor = jwt_processor
        self.settings = settings

        self.__session = session

    async def __call__(self, registration_data: SignUpRequest,
client_info: RequestClient) -> SuccessSignUpSchema:
            async with self.__session.begin():
                if await
self.user_service.exist_by_email(email=registration_data.email):
                    raise
UserAlreadyExist(email=registration_data.email)

                user = await self.user_service.create(
                    data=registration_data
                )
                client = await self.client_service.create(
                    user_id=user.id,
                )

                await self.role_service.add_role(user_id=user.id,
role="client")

                user.roles += ["user", "client"]
                access_token =
self.jwt_processor.generate_access_token(
                    UserBaseSchema(
                        id=user.id,
                        roles=user.roles,
                    ),
                    expire_time=self.settings.access_token_expire
                )
                user_session = await self.user_session_service.create(
                    user_id=user.id,
                    expire_time=self.settings.refresh_token_expire,
                    device_id=client_info.device_id
                )
                return SuccessSignUpSchema(
                    access_token=access_token,
                    refresh_token=user_session.token,
                    token_type="bearer",
                    user=UserSchema(
                        id=user.id,
                        email=user.email,
                        roles=user.roles,
                        password_hash=user.password_hash
                    )
                )
```

Класс SignUpUseCase получает все зависимости через конструктор: user_service, user_session_service, client_service, role_service – абстрактные

интерфейсы сервисов, jwt_processor – компонент для генерации токенов, settings – настройки приложения, session – объект асинхронной сессии базы данных.

Все зависимости объявлены через интерфейсы-порты, определенные в /application/ports/services как Protocol-классы. Это соответствует принципу инверсии зависимостей: сценарий зависит от абстракции, а не от конкретной реализации. Конкретные реализации портов находятся в инфраструктурном слое /infrastructure/services и подставляются контейнером Dishka автоматически с помощью аннотации FromDishka[SignUpUseCase].

Сценарий реализован как вызываемый объект через метод __call__, который принимает данные регистрации SignUpRequest и информацию о клиенте RequestClient.

Атомарность достигается за счет открытия транзакции async with self.__session.begin(). Все операции выполняются в рамках этой транзакции и либо применяются целиком, либо откатываются при ошибке.

Все порты сервисов и репозиториев находятся в прикладном слое в каталогах /application/ports/services и /application/ports/repositories. Каждый порт описывает контракт для работы с конкретной сущностью, например: UserServicePort – для пользователей, RoleServicePort – для ролей. Такое разделение соответствует принципу единой ответственности.

Порт UserServicePort:

```
class UserServicePort(Protocol):
    async def create(self, data: SignUpRequest) -> UserSchema: ...
    async def exist_by_email(self, email: str) -> bool: ...
    async def get_by_email(self, email: str) -> UserSchema | None:
...
    async def get_by_id(self, user_id: int) -> UserSchema | None:
...
```

Методы внутри порта объявлены без реализации – вместо тела метода указывается многоточие. Это означает, что порт лишь описывает, какие операции должны быть доступны и какие типы они принимают и возвращают, но не определяет как именно они выполняются.

После создания нужных роутеров, сценариев использования и портов необходимо сделать реализации портов сервисов и репозиториев. Все реализации

находятся в инфраструктурном слое в каталогах /infrastructure/services и /infrastructure/repositories. Каждый класс реализации наследуется от соответствующего интерфейса и должен иметь все методы из класса родителя.

Пример класса реализации UserService:

```
class UserService(UserServicePort):
    def __init__(self, repo: UserRepositoryPort):
        self.repo = repo

    async def create(self, data: SignUpRequest) -> UserSchema:
        return await self.repo.create(data)

    async def exist_by_email(self, email: str) -> bool:
        return await self.repo.exist_by_email(email)

    async def get_by_email(self, email: str) -> UserSchema | None:
        return await self.repo.get_by_email(email)

    async def get_by_id(self, user_id: int) -> UserSchema | None:
        return await self.repo.get_by_id(user_id)
```

Класс UserService наследуется от UserServicePort и принимает в конструктор объект UserRepositoryPort – репозиторий, отвечающий за работу с данными. Сам сервис не выполняет запросы к базе данных или внешнему API, а делегирует их репозиторию. Использование промежуточных сервисных классов между сценариями использования и репозиториями позволяет инкапсулировать дополнительную обработку данных, например кэширование или вспомогательные преобразования.

Не все классы реализаций работают с базой данных или любым другим источником данных – некоторые из них реализуют вспомогательную инфраструктурную логику. Например класс JWTProcessor, необходимый для генерации и валидации JWT-токенов.

Класс JWTProcessor:

```
class JWTProcessor(JWTProcessorPort):
    def __init__(self, settings: Settings):
        self.settings = settings

    def generate_access_token(self, user_data: UserBaseSchema,
expire_time: int) -> str:
        now =  datetime.now(timezone.utc)
        payload = {
            "iat": now,
```

```
            "exp": now + timedelta(seconds=expire_time),
            "sub": str(user_data.id),
            "roles": user_data.roles
        }
        encoded_jwt = jwt.encode(payload,
key=self.settings.jwt_private_key,
algorithm=self.settings.jwt_algorithm)
        return encoded_jwt

    def parse(self, access_token: str) -> UserBaseSchema:
        try:
            payload = jwt.decode(access_token,
self.settings.jwt_public_key,
algorithms=[self.settings.jwt_algorithm])
            user_id = payload.get("sub")
            if not user_id:
                raise ValueError("Access token missing 'sub' field")

            return UserBaseSchema(
                id=user_id,
                roles=payload.get("roles") or []
            )
        except jwt.ExpiredSignatureError:
            raise AccessTokenExpired
        except jwt.InvalidTokenError:
            raise AccessTokenInvalid
```

Порты репозиториев так же находятся в прикладном слое в каталоге /application/ports/repositories.

Класс UserRepositoryPort:

```
    class UserRepositoryPort(Protocol):
    async def create(self, data: SignUpRequest) -> UserSchema: ...
    async def exist_by_email(self, email: str) -> bool: ...
    async def get_by_email(self, email: str) -> UserSchema | None:
...
    async def get_by_id(self, user_id: int) -> UserSchema | None:
...
```

Реализации репозиториев находятся в инфраструктурном слое в каталоге /infrastructure/repositories и так же должны наследоваться от соответствующих классов портов и реализовывать все методы этих портов.

Репозитории отвечают за изоляцию логики доступа к данным от остальных слоев приложения. Репозиторий может работать с любым источником данных – базой данных или внешними API. В данном проекте все реализации репозиториев взаимодействуют с базой данных через асинхронную сессию SQLAlchemy,

предоставляя сервисам простой интерфейс без необходимости знать детали выполнения SQL-запросов.

Репозиторий UserRepository:

```
class UserRepository(UserRepositoryPort):
    def __init__(
        self,
        session: AsyncSession,
        password_processor: PasswordProcessorPort
    ):
        self.password_processor = password_processor
        self.__session = session

    async def create(self, data: SignUpRequest) -> UserSchema:
        password_hash =
self.password_processor.hash_password(data.password)
        user = User(
            first_name=data.first_name,
            last_name=data.last_name,
            middle_name=data.middle_name,
            email=data.email,
            password_hash=password_hash,
            is_active=True,
        )
        self.__session.add(user)
        await self.__session.flush()
        await self.__session.refresh(user)
        return UserSchema(
            id=user.id,
            roles=[],
            email=user.email,
            password_hash=user.password_hash
        )

    async def exist_by_email(self, email: str) -> bool:
        query = select(exists().where(User.email == email))
        result = await self.__session.execute(query)
        return result.scalar()

    async def get_by_email(self, email: str) -> UserSchema | None:
        query = (
            select(
                User.id,
                User.email,
                User.password_hash,
                func.array_remove(func.array_agg(Role.name),
None).label("roles")
            )
            .outerjoin(UserRole, User.id == UserRole.user_id)
            .outerjoin(Role, Role.id == UserRole.role_id)
            .where(User.email == email)
```

```
            .group_by(User.id)
        )
        result = await self.__session.execute(query)
        user = result.first()

        if user:
            return UserSchema(
                id=user.id,
                roles=user.roles,
                email=user.email,
                password_hash=user.password_hash,
            )

    async def get_by_id(self, user_id: int) -> UserSchema | None:
        query = (
            select(
                User.id,
                User.email,
                User.password_hash,
                func.array_remove(func.array_agg(Role.name),
None).label("roles")
            )
            .outerjoin(UserRole, User.id == UserRole.user_id)
            .outerjoin(Role, Role.id == UserRole.role_id)
            .where(User.id == user_id)
            .group_by(User.id)
        )
        result = await self.__session.execute(query)
        user = result.first()

        if user:
            return UserSchema(
                id=user_id,
                roles=user.roles,
                email=user.email,
                password_hash=user.password_hash,
            )
```

В конструктор класса UserRepository необходимо передать объект асинхронной сессии и сервис работы с паролями (хеширование и верификация).

Метод create принимает данные регистрации, хеширует пароль через объект password_processor, создает экземпляр ORM-модели и добавляет его в сессию. Затем вызывается flush – отправляется SQL-запрос в рамках текущей транзакции, не фиксируя ее, после чего вызывается refresh, чтобы получить актуальное состояние записи из базы данных – это необходимо, чтобы получить сгенерированный идентификатор пользователя (поле id таблицы User).

Для возможности отправлять запросы к базе данных, в файле /infrastructure/database.db.py создаются два объекта:

– async_engine – асинхронный движок, который отвечает за управление пулом соединений с базой данных;

– async_session_maker – фабрика сессий, основанная на асинхронном движке async_engine.

```
async_engine = create_async_engine(
    settings.DATABASE_URL_asyncpg,
    echo=False,
    future=True,
)
async_session_maker: async_sessionmaker[AsyncSession] =
async_sessionmaker(
    bind=async_engine,
    expire_on_commit=False
)
```

Фабрика не открывает соединения самостоятельно, она хранит настройки для их создания и в дальнейшем передается провайдеру Dishka, который выдает новую сессию при необходимости.

После написания всех реализаций необходимо создать провайдеры, которые свяжут интерфейсы с конкретными реализациями, чтобы внедрение зависимостей работало. Для этого в файле /infrastructure/di/providers.py созданы классы-наследники от Provider из библиотеки dishka и внутри этих классов описаны жизненные циклы объектов и правила сопоставления интерфейсов с конкретными реализациями.

Жизненный цикл определяется параметром scope и указывает, как долго существует созданный объект. В проекте используется два значения: scope.APP – объект создается один раз при запуске проекта и существует до его остановки, и scope.REQUEST – объект создается заново для каждого входящего HTTP-запроса и уничтожается после его завершения.

Рассмотрим провайдер SQLAlchemy сессии:

```
class SQLAlchemyProvider(Provider):
    @provide(scope=Scope.APP,
provides=async_sessionmaker[AsyncSession])
    def provide_session_maker(self) ->
async_sessionmaker[AsyncSession]:
```

```
        return async_session_maker

    @provide(scope=Scope.REQUEST, provides=AsyncSession)
    async def provide_session(self, session_maker:
async_sessionmaker[AsyncSession]) -> AsyncSession:
        async with session_maker() as session:
            yield session
```

Этот провайдер отвечает за предоставление сессий базы данных.

Метод provide_session_maker регистрируется с жизненным циклом scope.APP и возвращает фабрику сессий async_sessionmaker, которая создается один раз на все время работы приложения.

Метод provide_session регистрируется с жизненным циклом scope.REQUEST – для каждого запроса открывается новая асинхронная сессия.

Провайдеры репозиториев, сервисов и сценариев:

```
class ServicesProvider(Provider):
    scope = Scope.REQUEST

    user_service = provide(UserService, provides=UserServicePort)
    user_session_service = provide(UserSessionService,
provides=UserSessionServicePort)
    client_service = provide(ClientService,
provides=ClientServicePort)
    role_service = provide(RoleService, provides=RoleServicePort)
    catalog_service = provide(CatalogService,
provides=CatalogServicePort)
    content_service = provide(ContentService,
provides=ContentServicePort)
    schedule_service = provide(ScheduleService,
provides=ScheduleServicePort)
    trainer_service = provide(TrainerService,
provides=TrainerServicePort)
    enrollment_service = provide(EnrollmentService,
provides=EnrollmentServicePort)
    position_service = provide(PositionService,
provides=PositionServicePort)
    employee_service = provide(EmployeeService,
provides=EmployeeServicePort)

    jwt_processor = provide(JWTProcessor, provides=JWTProcessorPort)
    password_processor = provide(PasswordProcessor,
provides=PasswordProcessorPort)


class RepositoriesProvider(Provider):
    scope = Scope.REQUEST

    user_repo = provide(UserRepository, provides=UserRepositoryPort)
```

```
    user_session_repo = provide(UserSessionRepository,
provides=UserSessionRepositoryPort)
    client_repo = provide(ClientRepository,
provides=ClientRepositoryPort)
    role_repo = provide(RoleRepository, provides=RoleRepositoryPort)
    catalog_repo = provide(CatalogRepository,
provides=CatalogRepositoryPort)
    content_repo = provide(ContentRepository,
provides=ContentRepositoryPort)
    schedule_repo = provide(ScheduleRepository,
provides=ScheduleRepositoryPort)
    trainer_repo = provide(TrainerRepository,
provides=TrainerRepositoryPort)
    enrollment_repo = provide(EnrollmentRepository,
provides=EnrollmentRepositoryPort)
    position_repo = provide(PositionRepository,
provides=PositionRepositoryPort)
    employee_repo = provide(EmployeeRepository,
provides=EmployeeRepositoryPort)


class UseCasesProvider(Provider):
    scope = Scope.REQUEST

    login_uc = provide(LoginUseCase)
    logout_uc = provide(LogoutUseCase)
    logout_all_uc = provide(LogoutAllUseCase)
    refresh_tokens_uc = provide(RefreshTokensUseCase)
    signup_uc = provide(SignUpUseCase)

    create_position_uc = provide(CreatePositionUseCase)
    create_employee_uc = provide(CreateEmployeeUseCase)
    create_activity_uc = provide(CreateActivityUseCase)
    create_normative_uc = provide(CreateNormativeUseCase)
    create_course_uc = provide(CreateCourseUseCase)
    create_article_uc = provide(CreateArticleUseCase)
    update_article_uc = provide(UpdateArticleUseCase)
    create_weekly_slot_uc = provide(CreateWeeklySlotUseCase)
    duplicate_week_schedule_uc = provide(DuplicateWeekScheduleUseCase)
    delete_weekly_slot_uc = provide(DeleteWeeklySlotUseCase)
    create_trainer_uc = provide(CreateTrainerUseCase)
    upsert_questionnaire_uc = provide(UpsertQuestionnaireUseCase)
    enroll_on_course_uc = provide(EnrollOnCourseUseCase)
    update_trainer_profile_uc = provide(UpdateTrainerProfileUseCase)
    mark_attendance_uc = provide(MarkAttendanceUseCase)
    record_normative_results_uc =
provide(RecordNormativeResultsUseCase)
    mark_notification_read_uc = provide(MarkNotificationReadUseCase)
    mark_all_notifications_read_uc =
provide(MarkAllNotificationsReadUseCase)
```

Затем все провайдеры следует зарегистрировать в IoC-контейнере и написать функцию init_di, которая инициализирует его и свяжет с приложением FastAPI с помощью функции setup_dishka из библиотеки Dishka:

```
def ioc_factory() -> AsyncContainer:
    container = make_async_container(
        SQLAlchemyProvider(),
        ServicesProvider(),
        RepositoriesProvider(),
        UseCasesProvider(),
        SettingsProvider()
    )
    return container

def init_di(app: FastAPI) -> None:
    container = ioc_factory()
    setup_dishka(container, app=app)
```

Для возможности отслеживать ошибки, ход выполнения бизнес-сценариев и общее состояние приложения в проекте используется библиотека structlog. Ее настройка выполняется в файле /infrastructure/core/logger.py при старте приложения через функцию configure_logging:

```
def configure_logging() -> None:
    logging.basicConfig(
        format="%(message)s",
        stream=sys.stdout,
        level=logging.INFO,
    )

    structlog.configure(
        processors=[
            structlog.contextvars.merge_contextvars,
            structlog.processors.add_log_level,
            structlog.processors.format_exc_info,
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.JSONRenderer()
        ],



wrapper_class=structlog.make_filtering_bound_logger(logging.INFO),
        logger_factory=structlog.stdlib.LoggerFactory(),
        cache_logger_on_first_use=True,
    )

def get_logger(name: str | None = None) -> BoundLogger:
    return structlog.get_logger(name) if name else
structlog.get_logger()
```

Функция configure_logging настраивает два компонента. Сначала через logging.BasicConfig настраивается стандартный логгер Python: вывод направляется в stdout, минимальный уровень логирования INFO. Затем настраивается structlog – библиотека, расширяющая возможности стандартного логгера. Ключевой особенностью structlog является цепочка процессоров, обрабатывающих каждое событие перед выводом: merge_contextvars добавляет контекстные переменные, add_log_level добавляет уровень логирования, format_exc_info форматирует информацию об исключениях. TimeStamper добавляет временную метку в формате ISO, а JSONRenderer сериализует итоговое сообщение в JSON. В итоге каждая запись в логе представляет собой структурированный JSON-объект, что упрощает последующий анализ и сбор логов внешними системами мониторинга.

Для получения логгера в любом модуле используется функция get_logger, которой передается имя модуля через __name__, это позволяет идентифицировать источник каждой записи в логе. После получения логгера запись событий происходит через методы log.info, log.warning и так далее, в которые помимо текстового сообщения можно передавать любые именованные параметры – они будут добавлены в JSON-объект как дополнительные поля.

Настройки приложения определены в файле /infrastructure/core/settings.py. Для их загрузки используется библиотека pydantic-settings, которая автоматически считывает переменные окружения из файла .env. Базовый класс EnvBaseSettings наследуется от BaseSettings и задает общую конфигурацию.

Настройки разделены на классы, каждый из которых наследуется от EnvBaseSettings:

– FastAPISettings – содержит параметры веб-сервера: хост, порт, количество рабочих процессов.

– DBSettings – параметры подключения к базе данных, строка подключения к базе данных.

– JWTSettings – параметры JWT-аутентификации: алгоритм, публичный и приватный ключи, время жизни токенов.

– NatsBrokerSetttings – параметры подключения к NATS.

Итоговый класс Settings объединяет все группы через множественное наследование и дополняет их двумя свойствами:

– COOKIE_SETTINGS – возвращает словарь с параметрами cookie для refresh-токена.

– static_path – возвращает путь до папки с изображениями.

– coach_avatars_sub_dir – возвращает путь до папки с изображениями тренеров.

Файл settings.py:

```
class EnvBaseSettings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file=f"{env_path}/.env",
        extra="ignore",
    )


class FastAPISettings(EnvBaseSettings):
    DOMAIN_HOST: str
    DOMAIN_PORT: int | None = None
    WORKERS: int = 4


class NatsBrokerSettings(EnvBaseSettings):
    NATS_URL: str
    NATS_PORT: int

    # JetStream
    js_stream: str
    js_subject: str
    js_durable: str
    @property
    def NATS_BROKER_URL(self):
        return f"nats://{self.NATS_URL}:{self.NATS_PORT}"
class DBSettings(EnvBaseSettings):
    DB_HOST: str
    DB_PORT: int
    DB_USER: str
    DB_PASS: str
    DB_NAME: str

    @property
    def DATABASE_URL_asyncpg(self):
        return
f"postgresql+asyncpg://{self.DB_USER}:{self.DB_PASS}@{self.DB_HOST}:{s
elf.DB_PORT}/{self.DB_NAME}"
```

```
class JWTSettings(EnvBaseSettings):
    jwt_algorithm: str = "RS256"
    jwt_public_key: str
    jwt_private_key: str
    access_token_expire: int
    refresh_token_expire: int
    request_timeout_sec: int = 25


class Settings(
    NatsBrokerSettings,
    DBSettings,
    FastAPISettings,
    JWTSettings
):
    @property
    def COOKIE_SETTINGS(self):
        return {
            "key": "refresh_token",
            "httponly": True,
            "secure": True,
            "samesite": "lax",
            "max_age": self.refresh_token_expire
        }

    @property
    def static_path(self):
        return static_path

    @property
    def coach_avatars_sub_dir(self):
        return coach_avatars_sub_dir
```

Для удаления истекших сессий пользователей и рассылки уведомлений необходимо использовать фоновые задачи. Для этого в проекте используется асинхронная очередь Taskiq и брокер сообщений NATS.

Настройка компонентов выполняется в инфраструктурном слое в файле /infrastructure/taskiq/broker.py:

```
class ExtendedNATSKeyValueScheduleSource(NATSKeyValueScheduleSource):
    def __init__(
        self,
        *args,
        kv_config: Optional[KeyValueConfig] = None,
        **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.kv_config = kv_config

    async def startup(self) -> None:
```

```
        self.nats_client = await nats.connect(
            servers=self.servers,
            **self.connect_options,
        )
        self.nats_jetstream = self.nats_client.jetstream()

        try:
            self.kv = await
self.nats_jetstream.key_value(self.bucket_name)
        except BucketNotFoundError:
            if self.kv_config is not None:
                self.kv = await self.nats_jetstream.create_key_value(
                    config=self.kv_config
                )
            else:
                self.kv = await self.nats_jetstream.create_key_value(
                    bucket=self.bucket_name,
                )

broker = PullBasedJetStreamBroker(
    servers=[settings.NATS_BROKER_URL],
    durable="cleanup_worker",
    stream_name="background_tasks",
    subject="scheduled.cleanup_session",
    stream_config=StreamConfig(
        max_age=172800,
        retention=RetentionPolicy.WORK_QUEUE,
    )
)


source = ExtendedNATSKeyValueScheduleSource(
    servers=settings.NATS_BROKER_URL,
    kv_config=KeyValueConfig(
        bucket="taskiq_schedules",
        ttl=129600,
    ),
)

scheduler = TaskiqScheduler(
    broker=broker,
    sources=[source, LabelScheduleSource(broker)],
)

@broker.on_event(TaskiqEvents.WORKER_STARTUP)
async def startup(ctx) -> None:
    container = ioc_factory()
    setup_dishka(broker=broker, container=container)
```

Брокер PullBasedJetStreamBroker подключается к NATS и использует JetStream – расширение NATS, обеспечивающее персистентность сообщений и гарантий их доставки.

Брокеру задается имя стрима, subject (для маршрутизации сообщений) и durable (потребитель, который позволяет возобновлять обработку с того же места после перезапуска).

В конфигурации стрима указывается политика хранения WORK_QUEUE – сообщения удаляются из стрима сразу после успешной обработки, и максимальное время жизни сообщения в очереди – 2 дня.

Для хранения расписания периодических задач используется NATS Key-Value – встроенное хранилище типа ключ-значение. Поскольку стандартный класс NATSKeyValueScheduleSource из библиотеки Taskiq не поддерживает передачу настроек при создании KV-хранилища, был реализован класс-наследник ExtendedNATSKeyValueScheduleSource. Он принимает дополнительный параметр kv_config и при инициализации пытается подключиться к существующему хранилищу, а если оно не найдет, то создает новое с переданными настройками. В конфигурации KV-хранилища указывается имя бакета и время жизни записей ttl=129600 секунд.

Для запуска периодических задач по расписанию используется TaskiqScheduler. В качестве источников расписания используются ExtendedNATSKeyValueScheduleSource для динамических задач, хранящихся в NATS KV, и LabelScheduleSource для задач с расписанием, заданным через декоратор в коде – например через параметр cron.

При запуске воркера срабатывает событие WORKER_STARTUP, в обработчике которого инициализируется контейнер зависимостей Dishka и подключается к брокеру через setup_dishka. Это позволяет Taskiq-задачам получать зависимости.

Все задачи описываются в файле /infrastructure/taskiq/tasks.py.

Например задача проверять базу данных на наличие истекших сессий пользователей:

```
@broker.task(schedule=[{"cron": "*/30 * * * *"}])
@inject
async def expire_session(uc: FromDishka[ExpireSessionUseCase]):
    await uc(500)
```

Периодическая задача expire_session регистрируется через декоратор broker.task с параметром schedule, в котором задается cron-выражение на выполнение задачи каждые 30 минут. Декоратор @inject обеспечивает внедрение зависимостей через Dishka. При каждом запуске задача вызывается сценарий на удаление истекших сессий из базы данных.

Сборка приложения выполняется в файле /api/core/setup.py:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    await source.startup()
    if not broker.is_worker_process:
        await scheduler.startup()

    yield

    await source.shutdown()
    if not broker.is_worker_process:
        await scheduler.shutdown()


def register_routes(app: FastAPI):
    app.include_router(auth_router)
    app.include_router(courses_router)
    app.include_router(public_content_router)
    app.include_router(notifications_router)
    app.include_router(client_router)
    app.include_router(trainer_router)
    app.include_router(admin_router)
    app.include_router(admin_content_router)


def register_exception_handlers(app: FastAPI):
    app.add_exception_handler(Exception, unhandled_exception_handler)


def create_fastapi_app() -> FastAPI:
    app = FastAPI(
        title="Training Center API",
        version="1.0.0",
        docs_url="/docs",
        redoc_url="/redoc",
        openapi_url="/openapi.json",
        lifespan=lifespan,
    )
```

```
    init_di(app)
    register_routes(app)
    register_exception_handlers(app)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=["http://localhost:5173"],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )
    app.mount("/media", StaticFiles(directory=settings.static_path),
name="media")
    return app
```

Функция create_fastapi_app создает экземпляр FastAPI с базовыми параметрами: названием, версией и URL-адресами для документации Swagger и ReDoc. После создания приложения последовательно выполняется инициализация контейнера зависимостей Dishka через init_di, регистрация роутеров через register_routes – каждый роутер подключается через app.include_router. Также подключается middleware CORS, разрешающий запросы с фронтенда, и монтируется директория со статическими файлами по пути /media.

Жизненный цикл приложения управляется через asynccontextmanager-функцию lifespan, передаваемую в FastAPI при создании. При старте приложения инициализируется источник расписания source и планировщик scheduler.

Запуск веб-сервера выполняется через Uvicorn и настраивается в файле main.py:

```
app = create_fastapi_app()

def configure_webserver() -> uvicorn.Server:

    loop: Literal["auto", "uvloop"]
    if platform.system().lower() == "linux":
        try:
            import uvloop

            uvloop.install()
            loop = "uvloop"
        except ModuleNotFoundError:
            loop = "auto"
    else:
        loop = "auto"


    server = uvicorn.Server(
```

```
        uvicorn.Config(
            app,
            host=settings.DOMAIN_HOST,
            port=settings.DOMAIN_PORT,
            workers=settings.WORKERS,
            reload=False,
            forwarded_allow_ips="*",
            proxy_headers=True,
            server_header=False,
            date_header=False,
            log_level=logging.DEBUG,
            loop=loop,
        ),
    )
    return server

if __name__ == '__main__':
    server = configure_webserver()
    server.run()
```

Функция configure_webserver формирует конфигурацию сервера: хост, порт и количество рабочих процессов. Параметры proxy_headers и forwarded_allow_ips=”*” указывают серверу доверять заголовкам от обратного прокси (например Nginx), server_header=False и date_header-False отключают передачу заголовков с информацией о сервере в целях безопасности. Дополнительно выполняется проверка платформы: на Linux при наличии библиотеки uvloop она устанавливается в качестве event loop, что обеспечивает болеее высокую производительность по сравнению со стандартным asyncio event loop.

1.4.2 Реализация клиентской части приложения

1.4.2.1 Структура клиентской части и стек

Клиентская часть приложения реализована в виде SPA на основе библиотеки React с использованием языка TypeScript. В качестве инструмента сборки используется Vite. Для поддержки JSX и React требуется подключить плагин @vitejs/plugin-react, а также нужно указать порт сервера разработки [15].

Файл vite.config.ts:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
```

```
  server: {
    port: 5173,
  },
});
```

Адрес API указан в файле src/config.ts и читается из переменной окружения VITE_API_BASE_URL:

```
export const API_BASE_URL = import.meta.env.VITE_API_BASE_URL ??
'http://localhost:8000';
```

Исходный код организован следующим образом:

– В каталоге /src/api находятся модули для взаимодействия с API. Каждый модуль соответствует отдельной группе эндпоинтов: auth.ts, client.ts и так далее;

– Каталог /src/components содержит переиспользуемые компоненты интерфейса;

– В каталоге /stc/pages находятся страницы;

– Каталог /src/context содержит контекст аутентификации;

– Каталог /src/hooks содержит пользовательские хуки;

– В каталоге /src/lib хранятся вспомогательные утилиты;

– В каталоге /src/types хранятся TypeScript типы, описывающие структуры данных API.

Точкой входа приложения является файл /src/main.tsx, в котором React-дерево оборачивается в три провайдера: BrowserRouter – для клиентской маршрутизации через ReactRouter, AuthProvider – для управления состоянием аутентификации, React.StrictMode – необходим для выявления потенциальных проблем на этапе разработки: Итоговая структура приложения представлена на рисунке 19.

```
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <BrowserRouter>
      <AuthProvider>
        <App />
      </AuthProvider>
    </BrowserRouter>
  </React.StrictMode>,);
```

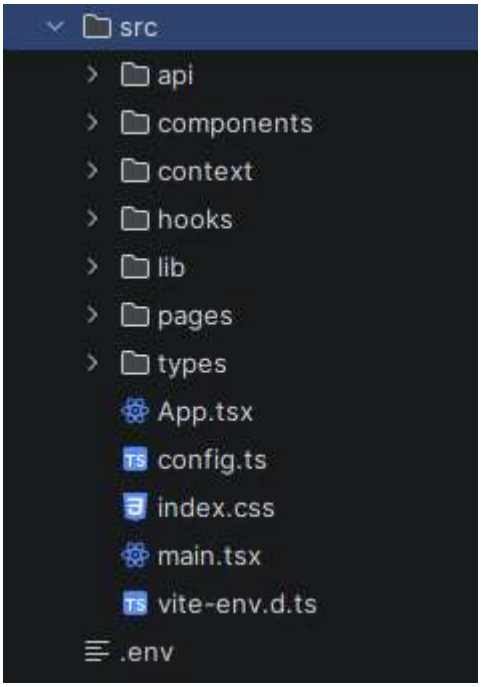


Рисунок 19 – Структура клиентской части приложения

1.4.2.2 Описание интерфейсов

Все TypeScript-интерфейсы, соответствующие структурам данных API, описываются в файле /src/types/api.ts. Они разделены на две группы: интерфейсы ответов и интерфейсы запросов. Интерфейсы запросов описывают данные, отправляемые на бэкенд:

```
export interface LoginRequest {
  email: string;
  password: string;
}

export interface SignUpRequest {
  email: string;
  password: string;
  first_name: string;
  last_name: string;
  middle_name?: string | null;
}
export interface EnrollCourseRequest {
  course_id: number;
  start_date?: string | null;
}
export interface DuplicateWeekScheduleRequest {
  source_week_start: string;
  target_week_start: string;
}
export interface RecordNormativeResultsRequest {
  client_id: number;
```

```
  notes?: string | null;
  results: NormativeValueInput[];
}
```

Интерфейсы ответов описывают структуру данных, возвращаемые бэкендом. Например, AuthResponse содержит access-токен и тип токена, а ClientProfileRead содержит полный набор полей профиля клиента:

```
export interface AuthResponse {
  access_token: string;
  token_type: string;
}
export interface ClientProfileRead {
  id: number;
  first_name: string;
  last_name: string;
  middle_name?: string | null;
  email: string;
  birth_date?: string | null;
  phone?: string | null;
  height_cm?: number | null;
  weight_kg?: number | null;
  emergency_contact_name?: string | null;
  emergency_contact_phone?: string | null;
  medical_notes?: string | null;
  active_courses: string[];
}
```

Отдельно описан тип Role и интерфейс JwtPayload, которые нужны для работы с JWT токеном на стороне клиента. Role представляет собой строковый тип с перечислением допустимых значений: client, trainer, admin. JwtPayload описывает структуру токена: идентификатор пользователя, список ролей, время истечения и выдачи токена:

```
export type Role = 'client' | 'trainer' | 'admin' | string;

export interface JwtPayload {
  sub: string;
  roles?: Role[];
  exp?: number;
  iat?: number;
}
```

1.4.2.3 Реализация API модулей

Взаимодействие с API реализовано в каталоге /src/api и разделено на два уровня: базовый HTTP-клиент в файле http.ts и отдельные модули для каждой группы эндпоинтов.

В основе HTTP-клиента лежит функция apiRequest, которая принимает путь эндпоинта и объект настроек RequestOptions. Настройки расширяют стандартный RequestInit двумя дополнительными параметрами:

– auth – указывает, нужно ли добавлять access-токен в заголовке Authorization;

– retryOnAuthError – разрешает повторить запрос после обновления токена.

При каждом запросе автоматически добавляется заголовок X-Device-ID с уникальным идентификатором устройства, а для запросов с телом, не являющимся FormData, автоматически выставляется Content-Type: application/json.

Функция apiRequest:

```
export async function apiRequest<T>(path: string, options:
RequestOptions = {}): Promise<T> {
  const { auth = false, retryOnAuthError = auth, headers, ...rest } =
options;

  const requestHeaders = new Headers(headers ?? {});
  requestHeaders.set('X-Device-ID', getDeviceId());

  if (auth && accessToken) {
    requestHeaders.set('Authorization', `Bearer ${accessToken}`);
  }

  if (!(rest.body instanceof FormData) &&
!requestHeaders.has('Content-Type') && rest.body) {
    requestHeaders.set('Content-Type', 'application/json');
  }

  const response = await fetch(`${API_BASE_URL}${path}`, {
    credentials: 'include',
    ...rest,
    headers: requestHeaders,
  });

  if (response.status === 401 && auth && retryOnAuthError) {
    const refreshedToken = await refreshAccessToken();
    if (refreshedToken) {
      return apiRequest<T>(path, { ...options, retryOnAuthError: false
});
    }
    unauthorizedHandler?.();
  }

  if (!response.ok) {
```

```
    let detail = 'Произошла ошибка';
    try {
      const payload = await parseResponse<{ detail?: string
}>(response);
      detail = typeof payload === 'string' ? payload : payload?.detail
?? detail;
    } catch {

    }
    throw new ApiError(response.status, detail);
  }

  return parseResponse<T>(response);
}
```

Access-токен хранится в модульной переменной accessToken внутри http.ts и недоступен за пределами модуля напрямую, для установки этого токена используется функция setAccessToken. Такой подход позволяет хранит токен в памяти, не сохраняя его в localStorage или sessionStorage, что защищает от XSS-атак.

При получении ответа 401 Unauthorized от бэкенда HTTP-клиент автоматически выполняет обновление сессии. Если запрос выполнялся с флагом auth: true и бэкенд вернул 401, клиент вызывает refreshAccessToken, которая обращается к обработчику обновления токена, зарегистрированному через configureApiAuth.

Функции configureApiAuth и refreshAccessToken:

```
export function configureApiAuth(options: {
  refresh: RefreshHandler;
  onUnauthorized: UnauthorizedHandler;
}) {
  refreshHandler = options.refresh;
  unauthorizedHandler = options.onUnauthorized;}
async function refreshAccessToken(): Promise<string | null> {
  if (!refreshHandler) {
    return null;
  }
  if (!refreshPromise) {
    refreshPromise = refreshHandler().finally(() => {
      refreshPromise = null;
    });
  }

  return refreshPromise;
}
```

Чтобы исключить одновременную отправку нескольких запросов на обновление токена, например если несколько запросов упали с 401 одновременно, используется переменная refreshPromise. Если обновление уже выполняется, все последующие вызовы ожидают результата того же промиса. После успешного обновления исходный запрос повторяется с флагом retryOnAuthError: false, чтобы исключить бесконечный цикл. Если обновление не удалось, то вызывается unauthorizedHandler, который сбрасывает состояние аутентификации и перенаправляет пользователя на страницу входа. При неуспешном ответе функция пытается извлечь поле detail из тела ответа и выбрасывает исключение ApiError с HTTP статусом и текстом ошибки. Это позволяет страницам единообразно обрабатывать ошибки API.

На основе apiRequest реализованы отдельные модули для каждой группы эндпоинтов. Каждая функция модуля соответствует одному эндпоинту, принимает типизированные параметры и возвращает типизированный результат. Например, модуль client.ts содержит функции для работы с профилем, анкетой, записями на курсы и расписанием клиента.

Файл /src/api/client.ts:

```
export function getClientProfile() {
  return apiRequest<ClientProfileRead>('/client/profile', { auth: true
});
}
export function getQuestionnaire() {
  return apiRequest<QuestionnaireRead>('/client/questionnaire', {
auth: true });
}
export function upsertQuestionnaire(payload: QuestionnaireUpsert) {
  return apiRequest<QuestionnaireRead>('/client/questionnaire', {
    method: 'PUT',
    auth: true,
    body: JSON.stringify(payload),
  });
}
export function createEnrollment(payload: EnrollCourseRequest) {
  return apiRequest<EnrollmentRead>('/client/enrollments', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}
```

```
export function listEnrollments() {
  return apiRequest<EnrollmentRead[]>('/client/enrollments', { auth:
true });
}
export function getWeekPlan(weekStart: string, enrollmentId?: number)
{
  const params = new URLSearchParams({ week_start: weekStart });
  if (enrollmentId) {
    params.set('enrollment_id', String(enrollmentId));
  }
  return apiRequest<PlanItemRead[]>(`/client/week-
plan?${params.toString()}`, { auth: true });
}
export function getNormatives(courseId?: number) {
  const query = courseId ? `?course_id=${courseId}` : '';
  return
apiRequest<NormativeHistoryPoint[]>(`/client/normatives${query}`, {
auth: true });
}
```

Модуль auth.ts выделен отдельно, функция refreshTokens реализована напрямую через fetch, обходя apiRequest. Это нужно потому, что запрос на обновление токена не должен сам инициализировать повторное обновление токена при получении 401, иначе возникнет бесконечный цикл. Запрос отправляется с параметром credentials: “include”, что позволяет браузеру автоматически приложить cookie с refresh-токеном.

Файл /str/api/auth.ts:

```
export async function refreshTokens(): Promise<AuthResponse | null> {
  if (refreshPromise) {
    return refreshPromise;
  }

  refreshPromise = fetch(`${API_BASE_URL}/auth/refresh`, {
    method: 'POST',
    credentials: 'include',
    headers: {
      'X-Device-ID': getDeviceId(),
    },
  })
    .then(async (response) => {
      if (response.status === 401) {
        return null;
      }

      if (!response.ok) {
        let detail = 'Не удалось обновить сессию';
        try {
```

```
          const payload = await response.json();
          if (typeof payload?.detail === 'string') {
            detail = payload.detail;
          }
        } catch {
          // ignore parse error
        }
        throw new Error(detail);
      }

      return parseRefreshResponse(response);
    })
    .finally(() => {
      refreshPromise = null;
    });

  return refreshPromise;
}
}
```

1.4.2.4 Управление состоянием аутентификации

Управление состоянием аутентификации реализовано с помощью React Context API в файле src/context/AuthContext.tsx.

Контекст предоставляет всем компонентам приложения единый доступ к данным текущего пользователя и методам аутентификации.

Интерфейс AuthContextValue описывает всё, что контекст предоставляет наружу: объект user с данными пользователя, accessToken, IsAuthenticated (флаг наличия сессии), isBootstrapping (флаг начальной инициализации), а также методы signIn, signUp, logout, hasRole и refreshSession.

Интерфейс AuthContextValue:

```
interface AuthContextValue {
  user: SessionUser | null;
  accessToken: string | null;
  isAuthenticated: boolean;
  isBootstrapping: boolean;
  signIn: (payload: LoginRequest) => Promise<void>;
  signUp: (payload: SignUpRequest) => Promise<void>;
  logout: () => Promise<void>;
  hasRole: (...roles: Role[]) => boolean;
  refreshSession: () => Promise<string | null>;
}
export const AuthContext = createContext<AuthContextValue |
null>(null);
```

Access-токен хранится в состоянии React (useState), а не в localStorage, для защиты от XSS-атак – JavaScript на странице не может прочитать токен напрямую. Refresh-токен хранится в httpOnly cookie и автоматически прикладывается браузером к каждому запросу на /auth/refresh.

Вспомогательная функция applyToken применяет токен в трех местах: устанавливает его в модульную переменную HTTP-клиента через setAccessToken, сохраняет в состояние React и извлекает из него данные пользователя через getUserFromToken. Такой подход гарантирует, что все три значения всегда синхронизированы.

Функция applyToken:

```
function applyToken(token: string | null, setToken: (value: string | null) => void, setUser: (value: SessionUser | null) => void) {
  setAccessToken(token);
  setToken(token);
  setUser(getUserFromToken(token));
}
```

Компонент AuthProvider хранит три состояния: accessTokenState – текущий access-токен, user – данные пользователя из токена, isBootstrapping – флаг начальной инициализации приложения.

При монтировании компонента выполняются два эффекта:

Первый регистрирует в HTTP-клиенте обработчики через configureApiAuth – функцию refreshSession для автоматического обновления токена при получении ответа 401, и clearSession для сброса сессии если обновление не удалось.

Второй эффект запускает начальную инициализацию – вызов refreshSession для восстановления сессии пользователя после перезагрузки страницы.

Метод refreshSession обращается к эндпоинту обновления токена и применяет полученный access-токен через applyToken. В случае ошибки вызывается clearSession. Метод logout отправляет запрос на API для инвалидации сессии и в блоке finily всегда вызывает clearSession – таким образом локальная сессия сбрасывается даже если запрос к API завершился ошибкой.

Метод hasRole реализован через useCallback и проверяет наличие хотя бы одной из переданных ролей у текущего пользователя через утилиту hasAnyRole из src/lib/jwt.ts.

Итоговое значение контекста формируется через useMemo.

Компонент AuthProvider:

```
export function AuthProvider({ children }: { children: ReactNode }) {
  const [accessTokenState, setAccessTokenState] = useState<string |
null>(null);
  const [user, setUser] = useState<SessionUser | null>(null);
  const [isBootstrapping, setIsBootstrapping] = useState(true);

  const clearSession = useCallback(() => {
    applyToken(null, setAccessTokenState, setUser);
  }, []);

  const refreshSession = useCallback(async () => {
    try {
      const response = await refreshTokens();
      const token = response?.access_token ?? null;
      applyToken(token, setAccessTokenState, setUser);
      return token;
    } catch {
      clearSession();
      return null;
    }
  }, [clearSession]);

  useEffect(() => {
    configureApiAuth({
      refresh: refreshSession,
      onUnauthorized: clearSession,
    });
  }, [refreshSession, clearSession]);

  useEffect(() => {
    let isMounted = true;

    void (async () => {
      try {
        await refreshSession();
      } finally {
        if (isMounted) {
          setIsBootstrapping(false);
        }
      }
    })();

    return () => {
      isMounted = false;
```

```
    };
  }, [refreshSession]);

  const signIn = useCallback(async (payload: LoginRequest) => {
    const response = await signInRequest(payload);
    applyToken(response.access_token, setAccessTokenState, setUser);
  }, []);

  const signUp = useCallback(async (payload: SignUpRequest) => {
    const response = await signUpRequest(payload);
    applyToken(response.access_token, setAccessTokenState, setUser);
  }, []);

  const logout = useCallback(async () => {
    try {
      if (accessTokenState) {
        await logoutRequest();
      }
    } finally {
      clearSession();
    }
  }, [accessTokenState, clearSession]);

  const hasRoleFn = useCallback(
    (...roles: Role[]) => {
      if (!user) return false;
      return hasAnyRole(user.roles, roles);
    },
    [user],
  );

  const value = useMemo<AuthContextValue>(
    () => ({
      user,
      accessToken: accessTokenState,
      isAuthenticated: Boolean(user),
      isBootstrapping,
      signIn,
      signUp,
      logout,
      hasRole: hasRoleFn,
      refreshSession,
    }),
    [user, accessTokenState, isBootstrapping, signIn, signUp, logout,
hasRoleFn, refreshSession],
  );

  return <AuthContext.Provider
value={value}>{children}</AuthContext.Provider>;
}
```

1.4.2.5 Переиспользуемые компоненты

Переиспользуемые компоненты – это части интерфейса, которые используются на нескольких страницах. Вместо дублирования одного и того же кода на каждой странице, он выносится в отдельный компонент. Все такие компоненты находятся в каталоге /src/components.

Компонент GlobalHeader (рисунок 20):

```
export function GlobalHeader() {
  const { isAuthenticated, logout, hasRole } = useAuth();
  return (
    <header className="site-header">
      <div className="site-header__inner">
        <Link to="/" className="site-header__brand">Центр подготовки
кадров</Link>
        <nav className="site-header__nav">
          <NavLink to="/" className={activeClass}>Главная</NavLink>
          <NavLink to="/courses"
className={activeClass}>Курсы</NavLink>
          {hasRole('client') ? <NavLink to="/app/client"
className={activeClass}>Кабинет клиента</NavLink> : null}
          {hasRole('trainer') ? <NavLink to="/app/trainer"
className={activeClass}>Кабинет тренера</NavLink> : null}
          {hasRole('admin') ? <NavLink to="/app/admin"
className={activeClass}>Кабинет администратора</NavLink> : null}
        </nav>
        <div className="site-header__actions">
          <NotificationBell authenticated={isAuthenticated} />
          {isAuthenticated ? (
            <button type="button" className="button button--ghost"
onClick={() => void logout()}>
              Выйти
            </button>
          ) : (
            <>
              <Link to="/login" className="button button--
ghost">Вход</Link>
              <Link to="/register"
className="button">Регистрация</Link>
            </>
          )}
        </div>
      </div>
    </header>
  );
}
```

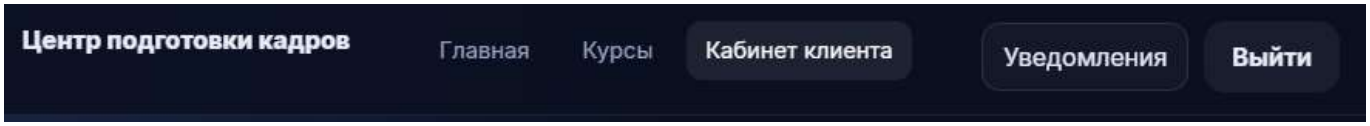


Рисунок 20 – Компонент GlobalHeader

Компонент GlobalHeader реализует шапку сайта, которая отображается на всех страницах. Он читает состояние аутентификации из контекста через хук useAuth и в зависимости от него динамически формирует навигацию: незарегистрированному пользователю показываются кнопки входа и регистрации, авторизованному – кнопка выхода и ссылки на личные кабинеты, соответствующие его ролям. Например, ссылка на кабинет администратора отображается только если у пользователя есть роль admin:

```
{hasRole('admin') ? <NavLink to="/app/admin"
className={activeClass}>Кабинет администратора</NavLink> : null}
```

Компонент AppShell:

```
export function AppShell() {
  const location = useLocation();
  const { user } = useAuth();
  const links = getCabinetLinks(location.pathname, user?.roles ?? []);

  return (
    <div className="app-layout">
      <GlobalHeader />
      <main className="app-layout__main">
        <div className="cabinet-tabs">
          {links.map((link) => (
            <NavLink key={link.to} to={link.to} end={link.to ===
'/app/client' || link.to === '/app/trainer' || link.to ===
'/app/admin'} className={({ isActive }) => isActive ? 'cabinet-tab
cabinet-tab--active' : 'cabinet-tab'}>
              {link.label}
            </NavLink>
          ))}
        </div>
        <div className="page-content">
          <Outlet />
        </div>
      </main>
    </div>
  );
}
```

Компонент AppShell (рисунок 21) реализует общую оболочку личных кабинетов. Он включает GlobalHeader и горизонтальную навигацию с вкладками, список которых формируется функцией getCabinetLinks на основе текущего маршрута и ролей пользователя.

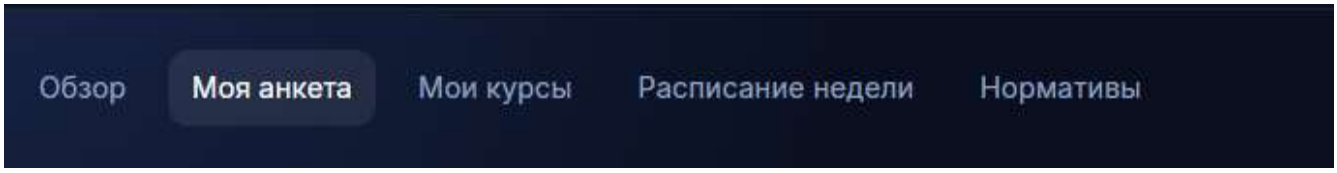


Рисунок 21 – Компонент AppShell

Функция getCabinetLinks:

```
function getCabinetLinks(pathname: string, roles: string[]) {
  if (pathname.startsWith('/app/admin')) {
    return [
      { to: '/app/admin', label: 'Обзор' },
      { to: '/app/admin/schedule', label: 'Расписание недели' },
      { to: '/app/admin/courses', label: 'Курсы' },
      { to: '/app/admin/activities', label: 'Активности' },
      { to: '/app/admin/normatives', label: 'Нормативы' },
      { to: '/app/admin/trainers', label: 'Тренеры' },
      { to: '/app/admin/articles', label: 'Статьи' },
    ];
  }
  if (pathname.startsWith('/app/trainer')) {
    return [
      { to: '/app/trainer', label: 'Обзор' },
      { to: '/app/trainer/schedule', label: 'Расписание по дням' },
      { to: '/app/trainer/profile', label: 'Мои данные' },
    ];
  }
  if (pathname.startsWith('/app/client')) {
    return [
      { to: '/app/client', label: 'Обзор' },
      { to: '/app/client/profile', label: 'Моя анкета' },
      { to: '/app/client/enrollments', label: 'Мои курсы' },
      { to: '/app/client/plan', label: 'Расписание недели' },
      { to: '/app/client/normatives', label: 'Нормативы' },
    ];
  }

  if (roles.includes('admin')) return [{ to: '/app/admin', label:
'Кабинет администратора' }];
  if (roles.includes('trainer')) return [{ to: '/app/trainer', label:
'Кабинет тренера' }];
  return [{ to: '/app/client', label: 'Кабинет клиента' }];
}
```

Например, находясь в кабинете клиента, пользователь видит вкладки «Обзор», «Моя анкета», «Мои курсы», «Расписание недели» и «Нормативы», а администратор в своем кабинете видит вкладки управления расписанием, курсами, тренерами и статьями. Содержимое текущей страницы рендерится через компонент Outlet из React Router, который подставляет дочерний маршрут внутрь оболочки.

Компонент ProtectedRoute:

```
export function ProtectedRoute({ roles }: { roles?: Role[] }) {
  const { isAuthenticated, isBootstrapping, hasRole } = useAuth();
  const location = useLocation();

  if (isBootstrapping) {
    return <div className="screen-center">Проверяем сессию…</div>;
  }

  if (!isAuthenticated) {
    return <Navigate to="/login" replace state={{ from: location }}
/>;
  }


  if (roles?.length && !hasRole(...roles)) {
    return <Navigate to="/app" replace />;
  }


  return <Outlet />;
}
```

Этот компонент отвечает за защиту маршрутов от неавторизованного доступа. Он принимает необязательный параметр roles – список ролей, которым разрешен доступ к маршруту. Логика работы следующая: пока выполняется начальная инициализация сессии, отображается надпись «Проверяем сессию…», если пользователь не авторизован, то выполняется перенаправление на страницу входа с сохранением исходного маршрута в состоянии навигации, если пользователь авторизован, но не имеет нужно роли, то перенаправляется на общую страницу, в противном случае рендерится дочерний маршрут через Outlet.

Таким образом реализованы следующие компоненты: EmptyState, NotificationBell, SectionCard, StartCard.

1.4.2.6 Реализация страниц

Страницы приложения находятся в каталоге /src/pages и разделены на следующие группы:

– auth – страницы входа и регистрации;

– public – публично доступные страницы;

– client, trainer, admin – страницы личных кабинетов.

Каждая страница представляет собой React-компонент, который при монтировании загружает необходимые данные через API-модули и отображает их пользователю.

Страницы LoginPage и RegisterPage доступны без авторизации. LoginPage хранит в локальном состоянии значения полей формы, флаг загрузки и текст ошибки. При отправке формы вызывается метод signIn из контекста аутентификации, после чего пользователь перенаправляется на страницу, с которой он пришел, либо на /app. Если пользователь уже авторизован и открывает страницу входа, то он сразу перенаправляется в нужный кабинет в зависимости от роли.

Страница LoginPage (рисунок 22):

```
  return (
    <div className="auth-page">
      <form className="auth-card" onSubmit={handleSubmit}>
        <div>
          <div className="badge">Авторизация</div>
          <h1>Вход в систему</h1>
        </div>

        {error ? <div className="alert alert--error">{error}</div> :
null}

        <label className="field">
          <span>Email</span>
          <input type="email" value={email} onChange={(event) =>
setEmail(event.target.value)} required />
        </label>
        <label className="field">
          <span>Пароль</span>
          <input type="password" value={password} onChange={(event) =>
setPassword(event.target.value)} required />
        </label>

        <button className="button" type="submit" disabled={loading}>
```

```
          {loading ? 'Входим…' : 'Войти'}
        </button>

        <div className="auth-links">
          <Link to="/register">Нет аккаунта? Зарегистрироваться</Link>
          <Link to="/courses">Смотреть курсы</Link>
        </div>
      </form>
    </div>
  );
}
```

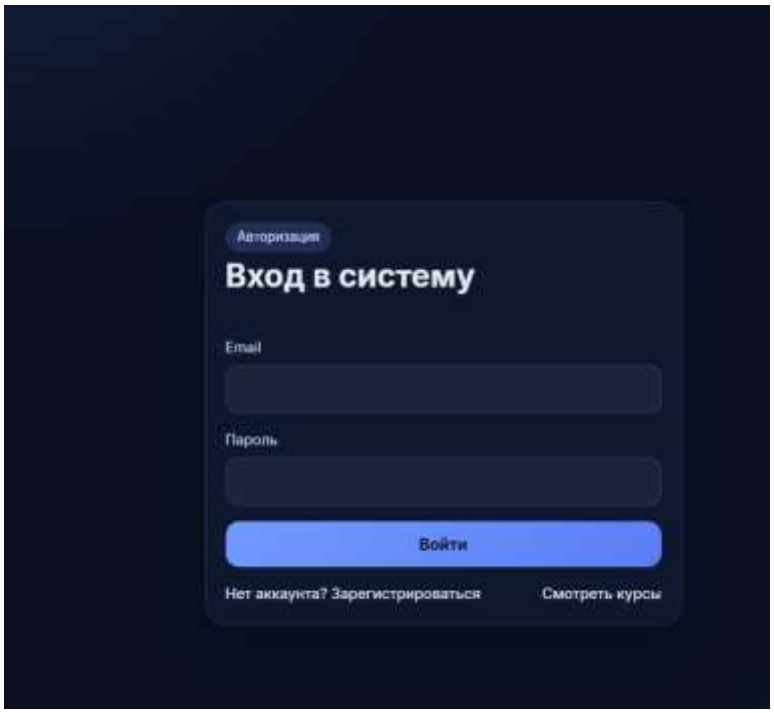


Рисунок 22 – Страница LoginPage

Публичные страницы (HomePage, PublicCoursesPage и др.) доступны всем пользователям без авторизации и отображают общую информацию о центре, курсах и статьях. На рисунке 23 представлена страница HomePage.

Страница ArticlePage:

```
  return (
    <div className="public-page">
      <GlobalHeader />
      <div className="page-content page-content--public">
        {error ? <div className="alert alert--error">{error}</div> :
null}
        {!article ? <div>Загрузка…</div> : (
          <article className="card article-view">
            <div className="card__body">
              <Link to="/" className="muted">← На главную</Link>
```

```
                <div className="muted">{formatDate(article.published_at
?? article.created)}</div>
                <h1>{article.title}</h1>
                {article.summary ? <p className="article-
view__summary">{article.summary}</p> : null}
                <div className="article-
view__content">{article.content}</div>
              </div>
            </article>
          )}
        </div>
      </div>
  );
}
```

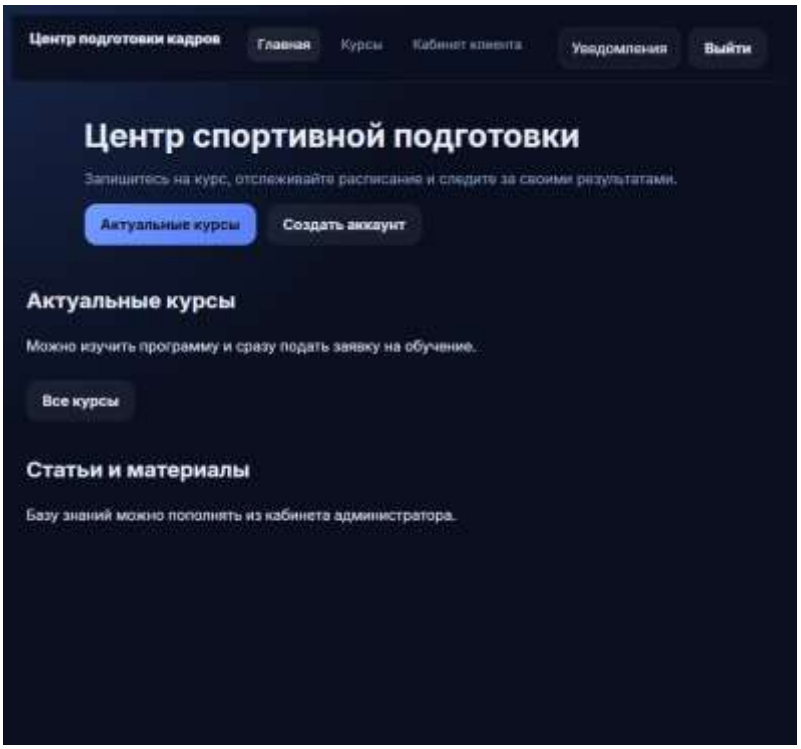


Рисунок 23 – Страница HomePage

Страницы кабинета клиента содержат пять разделов:

– ClientDashboardPage – главная страница кабинета, которая при монтировании параллельно запрашивает профиль, список зачислений и расписание на текущую неделю через Promise.all, после чего отображает сводную статистику и ближайшие занятия.

– ClientProfilePage загружает профиль и анкету через Promise.allSettled. Она содержит форму редактирования анкеты с медицинскими данными, которая при отправке вызывает upsertQuestionnaire и перезагружает данные.

– ClientEnrollemtPage отображает список записей на курсы.

– ClientPlanPage показывает расписание занятий на выбранную неделю.

– ClientNormativesPage отображает историю нормативов.

Таким образом сделаны все необходимые страницы сайта.

1.4.2.7 Маршрутизация

Маршрутизация приложения описана в файле src/app.tsx с использованием библиотеки React Router. Все маршруты объявлены декларативно внутри компонента Routes и разделены на три уровня:

– первый уровень – публичные маршруты, доступные без авторизации: главная страница «/», список курсов «/courses», страница отдельного курса «/courses/:courseId», а также страницы входа «/login» и регистрации «/register».

– второй уровень – защищенные маршруты, обернутые в ProtectedRoute без указания ролей. Сюда входят все маршруты под префиксом /app, внутри которых рендерится AppShel с навигацией. При переходе на /app без уточнения раздела срабатывает компонент DashboardRedirect, который перенаправляет пользователя в нужный кабинет в зависимости от его роли: например, администратор попадает на «/app/admin».

– третий уровень – маршруты кабинета администратора, которые обернуты в дополнительный ProtectedRoute с параметром roles={[“admin”]}. Это означает, что все маршруты «/app/admin/*» доступны только пользователям с ролью admin.

Файл App.tsx:

```
function DashboardRedirect() {
  const { user } = useAuth();
  if (user?.roles.includes('admin')) return <Navigate to="/app/admin"
replace />;
  if (user?.roles.includes('trainer')) return <Navigate
to="/app/trainer" replace />;
  return <Navigate to="/app/client" replace />;
}
function NotFoundPage() {
  return <div className="screen-center">Страница не найдена.</div>;
}
export default function App() {
  return (
```

```
    <Routes>
      <Route path="/" element={<HomePage />} />
      <Route path="/courses" element={<PublicCoursesPage />} />
      <Route path="/courses/:courseId" element={<PublicCoursePage />}
/>
      <Route element={<ProtectedRoute />}>
        <Route path="/app" element={<AppShell />}>
          <Route index element={<DashboardRedirect />} />
          <Route path="client/enrollments"
element={<ClientEnrollmentsPage />} />
          <Route path="client/plan" element={<ClientPlanPage />} />
          <Route path="client/normatives"
element={<ClientNormativesPage />} />
          <Route path="trainer" element={<TrainerDashboardPage />} />
          <Route path="trainer/profile" element={<TrainerProfilePage
/>} />
          <Route path="trainer/schedule" element={<TrainerSlotsPage
/>} />
          <Route element={<ProtectedRoute roles={['admin']} />}>
            <Route path="admin" element={<AdminDashboardPage />} />
            <Route path="admin/schedule"
element={<AdminWeeklySchedulePage />} />
            <Route path="admin/activities"
element={<AdminActivitiesPage />} />
            <Route path="admin/normatives"
element={<AdminNormativesPage />} />
            <Route path="admin/courses" element={<AdminCoursesPage />}
/>
            <Route path="admin/trainers" element={<AdminTrainersPage
/>} />
            <Route path="admin/articles" element={<AdminArticlesPage
/>} />
          </Route>
        </Route>
      </Route>

      <Route path="*" element={<NotFoundPage />} />
    </Routes>
  );
}
```

1.4.3 Тестирование и отладка

Для проверки корректности работы разработанного веб-приложения применены два взаимодополняющих подхода: автоматическое тестирование программного кода и ручное тестирование пользовательских сценариев с проверкой обработки ошибок.

Для проверки серверной части реализованы автоматические тесты на фреймворке pytest, размещённые в каталоге backend/tests/. Тесты покрывают

ключевые сценарии: регистрацию и авторизацию пользователей с проверкой выдачи токенов, обновление и инвалидацию сессий, работу с профилем клиента и анкетой, запись на курс и формирование персонального расписания, управление курсами и активностями со стороны администратора, назначение ролей и создание тренера, а также корректность работы эндпоинтов с недостаточными правами доступа. Запуск тестов выполняется командой pytest [16].

В ходе разработки автоматические тесты регулярно запускались после внесения изменений в код. Результаты ручного тестирования по наиболее распространённым пользовательским сценариям представлены в таблице 28.

Таблица 28 – Тест–кейсы ручного тестирования наиболее распространённых пользовательских ошибок

№	Наименование	Описание	Шаги для воспроизведения	Ожидаемый результат
1	Регистрация с пустыми обязательными полями	При регистрации пользователь должен заполнить все поля формы. Пропуск обязательных полей является одной из наиболее частых ошибок при работе с веб-формами	1. Открыть страницу регистрации. 2. Оставить одно из обязательных полей пустым. 3. Нажать кнопку «Зарегистрироваться»	Форма не отправляется на сервер; пользователю отображается сообщение об обязательности заполнения поля
2	Доступ к кабинету администратора без соответствующей роли	Страницы кабинета администратора должны быть недоступны пользователям без роли admin	1. Авторизоваться как клиент. 2. Попытаться открыть адрес /app/admin	Пользователь перенаправляется на страницу /app; доступ к кабинету администратора не предоставляется
3	Запись на курс без авторизации	Запись на курс доступна только авторизованным пользователям с ролью клиента. Попытка записаться без входа в систему должна корректно обрабатываться	1. Открыть страницу курса без авторизации. 2. Нажать кнопку «Записаться на курс»	Пользователь перенаправляется на страницу входа; запись на курс не выполняется

№	Наименование	Описание	Шаги для воспроизведения	Ожидаемый результат
4	Вход с неверным паролем	Для входа в систему требуются корректные учётные данные. Ввод неверного пароля является наиболее распространённой проблемой при авторизации	1. Открыть страницу входа. 2. Ввести существующий адрес электронной почты. 3. Ввести неверный пароль. 4. Нажать кнопку «Войти»	На экране отображается сообщение об ошибке авторизации; пользователь остаётся на странице входа и может повторить попытку
5	Регистрация с уже занятым адресом электронной почты	Адрес электронной почты должен быть уникальным в системе. Пользователи забывают о существующей учётной записи и пытаются зарегистрироваться повторно	1. Открыть страницу регистрации. 2. Ввести адрес электронной почты уже существующего пользователя. 3. Заполнить остальные поля. 4. Нажать кнопку «Зарегистрироваться»	На экране отображается сообщение о том, что пользователь с таким email уже существует; учётная запись не создаётся

1.4.4 Развертывание приложения

1.4.4.1 Контейнеризация

Для развертывания серверной и клиентской частей приложения используется контейнеризация на основе Docker, обеспечивающая независимость от конкретной операционной системы и конфигурации хоста [17].

Контейнеризация серверной части:

Dockerfile серверной части использует образ python:3.12-slim. Переменные PYTHONUNBUFFERED=1 и PYTHONDONTWRITEBYTECODE=1 нужны, чтобы вывод логов был без буфферизации и для отключения создания файлов __pycache__. Далее копируется и устанавливается файл зависимостей requirements.txt, после чего копируется весь исходный код. Такой порядок позволяет использовать кэш Docker – при повторной сборке без изменений в зависимостях слой с pip install не пересобирается.

Docker-compose файл описывает четыре сервиса: api, worker, scheduler и nats. Первые три сервиса на основе Python собираются из одного Dockerfile, но запускаются с разными командами и в разных потоках. Переменные окружения у сервисов читаются из файла .env через env_file. Все сервисы (кроме самого nats)

зависят от nats с условием service_healthy. Это гарантирует, что NATS успеет полностью запуститься до того, как к нему попытаются подключиться остальные сервисы. Данные NATS сохраняются в nats-data, что обеспечивает их сохранность при перезапуске контейнера.

Файл Dockerfile:

```
FROM python:3.12-slim

ENV PYTHONUNBUFFERED=1
ENV PYTHONDONTWRITEBYTECODE=1

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
```

Файл docker-compose.yaml:

```
name: backend-training-centre

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    env_file:
      - .env
    ports:
      - "${DOMAIN_PORT}:${DOMAIN_PORT}"
    environment:
      - TZ=UTC
    depends_on:
      nats:
        condition: service_healthy

    command: >
      python -m main


  nats:
    image: nats:2.10-alpine
    healthcheck:
      test: [ "CMD", "nats-server", "--help" ]
      interval: 5s
      timeout: 3s
      retries: 5
    command: ["-js", "-m", "8222", "-V", "--store_dir", "/data"]
```

```
    environment:
      - TZ=UTC
    volumes:
      - nats-data:/data
  worker:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - TZ=UTC
    command: >
      taskiq worker src.infrastructure.taskiq.broker:broker --fs-
discover -tp src/infrastructure/taskiq/tasks.py --workers 2 --ack-type
when_saved
    env_file:
      - .env
    depends_on:
      nats:
        condition: service_healthy



  scheduler:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - TZ=UTC
    command: >
      taskiq scheduler src.infrastructure.taskiq.broker:scheduler --
fs-discover -tp src/infrastructure/taskiq/tasks.py
    env_file:
      - .env
    depends_on:
      nats:
        condition: service_healthy



volumes:
  nats-data:
```

Для запуска серверной части сначала необходимо создать файл .env и заполнить необходимыми переменными. Затем выполняется сборка и запуск контейнеров:

```
docker compose up --build -d
```

Флаг --build пересобирает образы при наличии изменений в коде, а флаг -d запускает контейнеры в фоновом режиме.

Контейнеризация клиентской части:

Dockerfile клиентской содержит несколько этапов. На первом этапе используется образ node:20-alpine, указывается рабочая директория, копируется файл package.json и затем устанавливается. После этого копируется весь исходный код, передается переменная сборки VITE_API_BASE_URL через ARG и выполняется команда npm run build, которая компилирует TypeScript и собирает бандл. На втором этапе используется образ nginx:alpine, далее копируется только собранный бандл из директории dist, а также файл конфигурации nginx.

Docker compose клиентской части содержит один сервис frontend. Сервис собирается из Dockerfile и принимает переменную сборки VITE_API_URL через args – Docker compose подставляет ее значение из файла .env автоматически. Порт 80 контейнера, который слушает Nginx, проброшен на порт 5173.

Файл Dockerfile:

```
FROM node:20-alpine AS builder

WORKDIR /app

COPY package.json .
RUN npm install

COPY . .

ARG VITE_API_BASE_URL
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80
```

Файл docker-compose.yaml:

```
name: frontend-training-centre

services:
  frontend:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        - VITE_API_BASE_URL=${VITE_API_BASE_URL}
```

```
    ports:
      - "5173:80"
```

Файл nginx.conf:

```
server {
    listen 80;

    location / {
        root /usr/share/nginx/html;
        index index.html;
        try_files $uri $uri/ /index.html;
    }
}
```

1.4.4.2 Настройка сервера

Для развертывания приложения на сервере была выполнена его первоначальная настройка. В качестве операционной системы используется Ubuntu.

Первым шагом выполняется обновление системных пакетов:

```
sudo apt update && sudo apt full-upgrade -y
```

Затем создается новый пользователь и добавляется в группу sudo, что дает ему права на выполнение команд от имени администратора:

```
adduser tc
usermod -aG sudo tc
```

В файле /etc/sudoerc через команду sudo visudo добавляется строка, которая разрешает пользователю выполнять любые команды с повышенными привелегиями:

```
tc ALL=(ALL:ALL) ALL
```

Для ограничения доступа к серверу нужно настроить аутентификацию по SSH-ключу. Для этого сначала нужно сгенерировать пару ключей на локальном устройстве:

```
ssh-keygen -t ed25519 -C "tc@windows"
```

Затем полученный публичный ключ переносится на сервер:

```
ssh-copy-id -i ~/.ssh/training_centre_key.pub tc@host
```

После проверки подключения по ключу в конфигурационном файле /etc/ssh/sshd_config отключается аутентификация по паролю и запрещается вход от имени root:

```
PasswordAuthentication no
PermitRootLogin no
```

После этого нужно перезапустить SSH службу, чтобы изменения применились:

```
systemctl restart sshd
```

Далее настраивается UFW. Устанавливается политика по умолчанию – блокировать все входящие соединения и разрешать всех исходящие. Затем явно разрешается только SSH-подключение на порту 22:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow 22/tcp comment "ssh"
sudo ufw enable
```

Затем настраивается база данных. Сначала ее необходимо установить командой:

```
sudo apt install postgresql postgresql-contrib -y
```

После нужно подключиться от имени postgres и создать нового пользователя и базу данных:

```
sudo -u postgres psql
```

```
CREATE USER tc_bd WITH PASSWORD 'xxxx';
CREATE DATABASE training_centre OWNER tc_bd;
GRANT ALL PRIVILEGES ON DATABASE training_centre TO tc_bd;
```

1.4.4.3 Перенос проекта на сервер

Для переноса исходного кода на сервер используется платформа GitHub. Сначала были инициализированы репозитории на локальном устройстве командой git init. Затем с помощью команд git add . и git commit -m “init proj” были добавлены и зафиксированы все файлы проекта. После этого созданы удаленные репозитории на GitHub, на которые были загружены файлы проекта командой git push [18].

Для клонирования репозиториев с сервера без пароля и для безопасности настраивается аутентификация по SSH-ключу. На сервере генерируется пара ключей:

```
ssh-keygen -t ed25519 -C "server@training-centre"
```

Полученный публичный ключ добавляется в настройки GitHub. После чего можно клонировать репозитории:

git@github.com:Sergey-xxxx/tc_diplom_backend.git

После переноса проекта нужно настроить окружения серверной и клиентской частей.

Для настройки серверной части в первую очередь нужно создать файл .env. Для этого копируются переменные из файла .env.example в файл .env:

```
cp .env.example .env
```

После чего файл .env нужно открыть заполнить нужными значениями:

```
nano .env
```

Для настройки клиентской части также создается .env файл с переменной VITE_API_URL, указывающий на ардес API.

Для обеспечения доступа к сайту по доменному имени устанавливается и настраивается Nginx:

```
sudo apt install nginx -y
sudo systemctl enable nginx
sudo systemctl start nginx
```

Затем создается конфигурационный файл для домена:

```
sudo nano /etc/nginx/sites-available/training-centre
```

На начальном этапе настраивается только для HTTP:

```
server {
    listen 80;
    server_name domain.com www.domain.com;

    location / {
        proxy_pass http://127.0.0.1:5173;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /api/ {
```

```
            proxy_pass http://127.0.0.1:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
```

Конфиг активируется и Nginx перезапускается:

```
sudo ln -s /etc/nginx/sites-available/training-centre
/etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

В UFW открываются порты для HTTP и HTTPS:

```
sudo ufw allow 80/tcp comment "http"
sudo ufw allow 443/tcp comment "https"
```

Чтобы получить SSL-сертификат нужно установить Certbot:

```
sudo apt install certbot python3-certbot-nginx -y
sudo certbot --nginx -d domain.com -d www.domain.com
```

Certbot автоматически редактирует конфиг Nginx, добавляя HTTPS и редирект с HTTP на HTTPS. Сертификаты Lets Encrypt выдаются на 90 дней и обновляются автоматически через systemd таймер.

После завершения настройки Nginx запускаются контейнеры серверной и клиентской частей:

```
docker compose up --build -d
```

Затем применяются миграции базы данных:

```
docker compose exec api alembic upgrade head
```

После выполнения всех шагов приложение доступно по адресу https://f-348-f-89374-hf-sf-8374.online.

1.4.5 Руководство пользователя

Настоящее руководство описывает основные сценарии работы с веб-приложением центра подготовки кадров – от регистрации новой учётной записи до использования функций личного кабинета.

Для начала работы с системой пользователь открывает главную страницу приложения, где представлены актуальные курсы и статьи центра. Для получения

доступа к личному кабинету необходимо перейти по ссылке «Зарегистрироваться» в шапке сайта.

В форме регистрации необходимо заполнить следующие поля: фамилия, имя, отчество (необязательно), адрес электронной почты и пароль. После заполнения всех обязательных полей следует нажать кнопку «Зарегистрироваться». При успешной регистрации пользователь автоматически перенаправляется в личный кабинет клиента. Форма регистрации представлена на рисунке 24. Вход в систему представлен на рисунке 25.

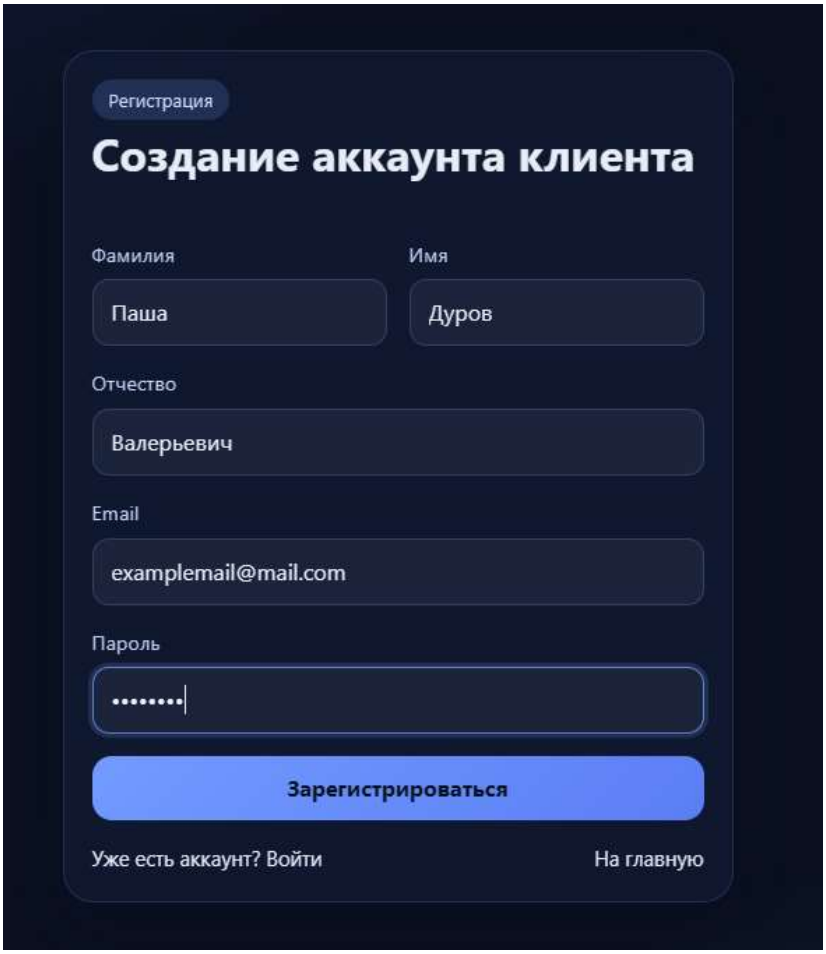


Рисунок 24 – Форма регистрации

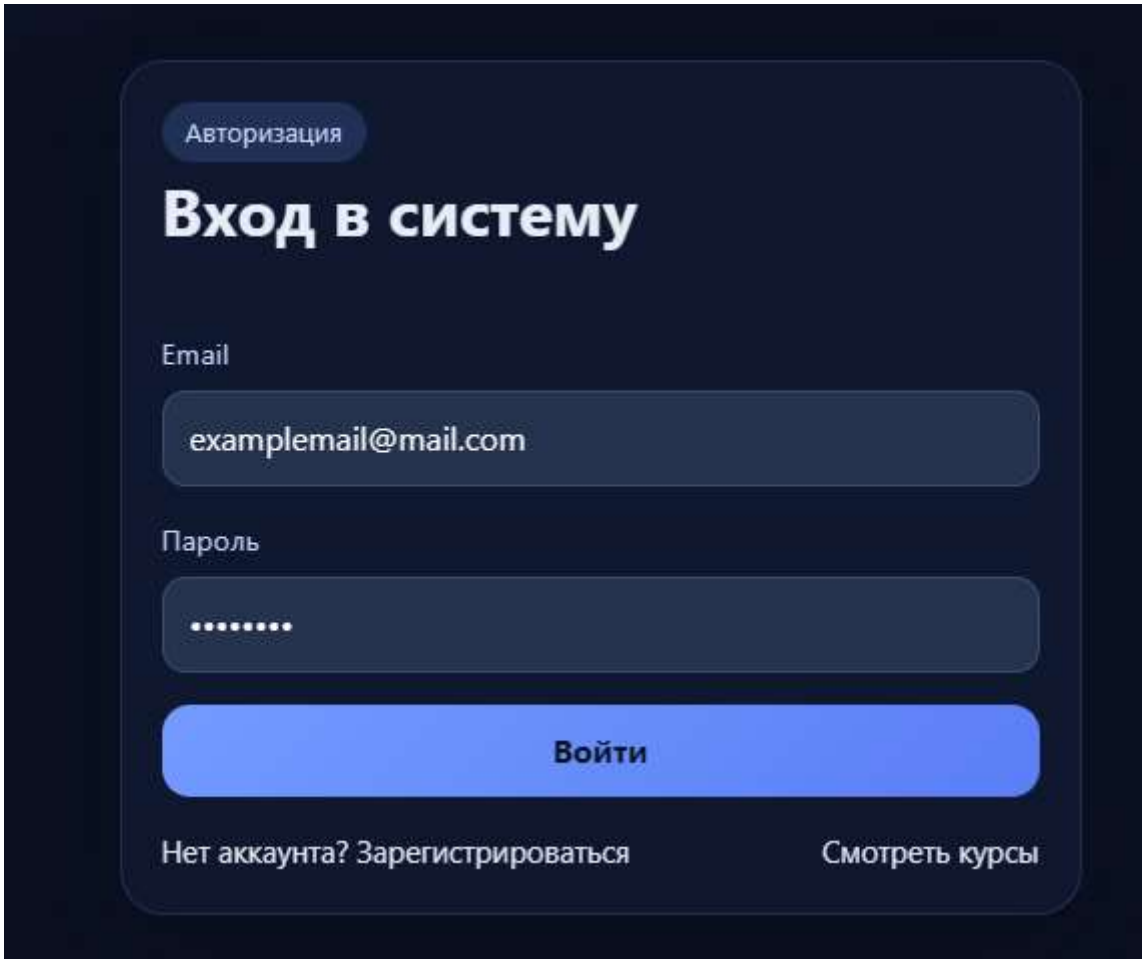


Рисунок 25 – Форма входа в систему

После входа в систему клиент попадает в личный кабинет, где доступны следующие разделы: обзор активных курсов и ближайших занятий, анкета с личными и медицинскими данными, список зачислений на курсы, недельное расписание и динамика нормативов. Личный кабинет клиента представлен на рисунке 26:

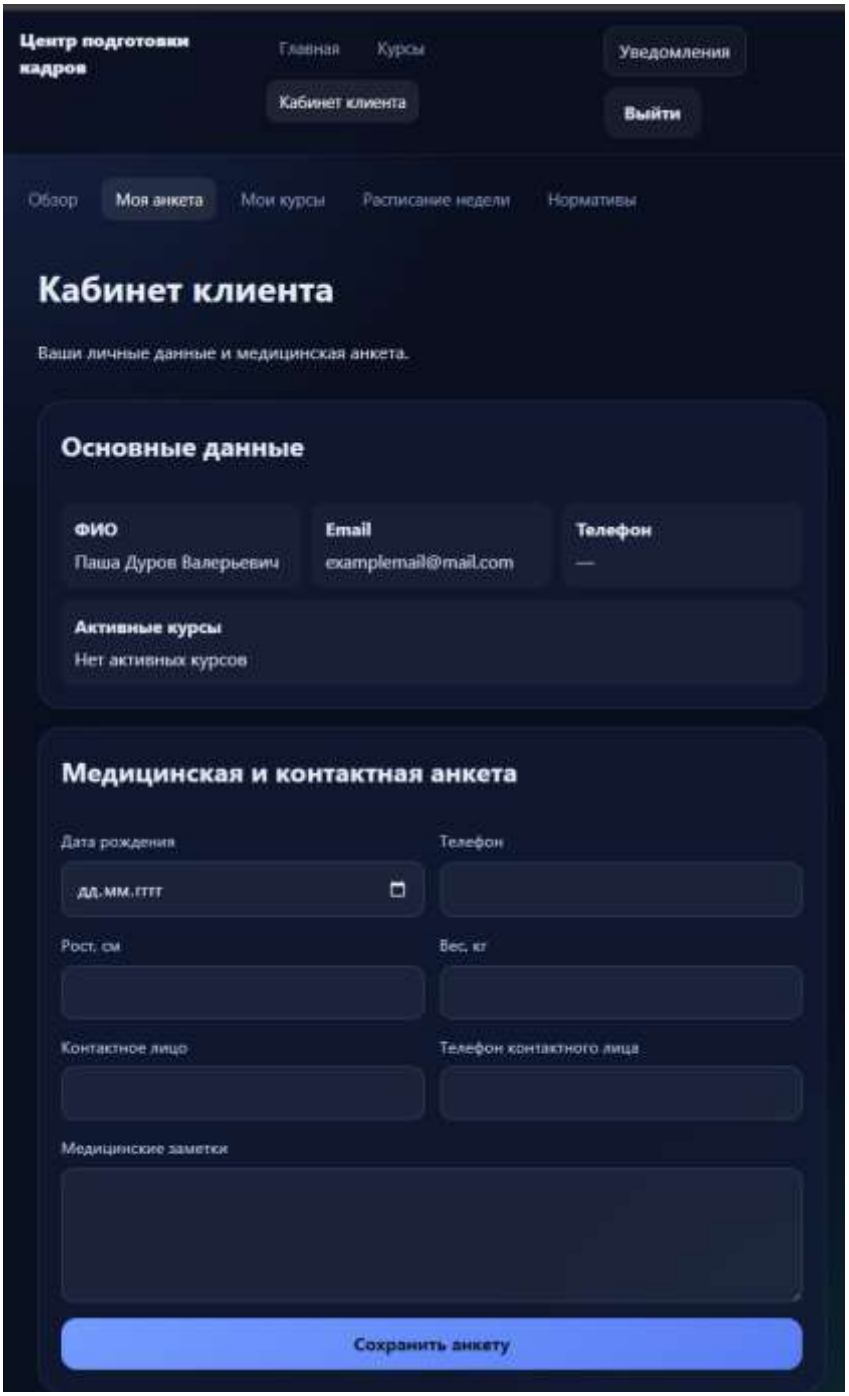


Рисунок 26 – Личный кабинет клиента

Для записи на курс необходимо перейти в раздел «Курсы», выбрать интересующий курс, ознакомиться с программой занятий и нажать кнопку «Записаться на курс», указав желаемую дату начала. Страница курсов изображена на рисунке 27, а подробная информация о курсе представлена на рисунке 28.

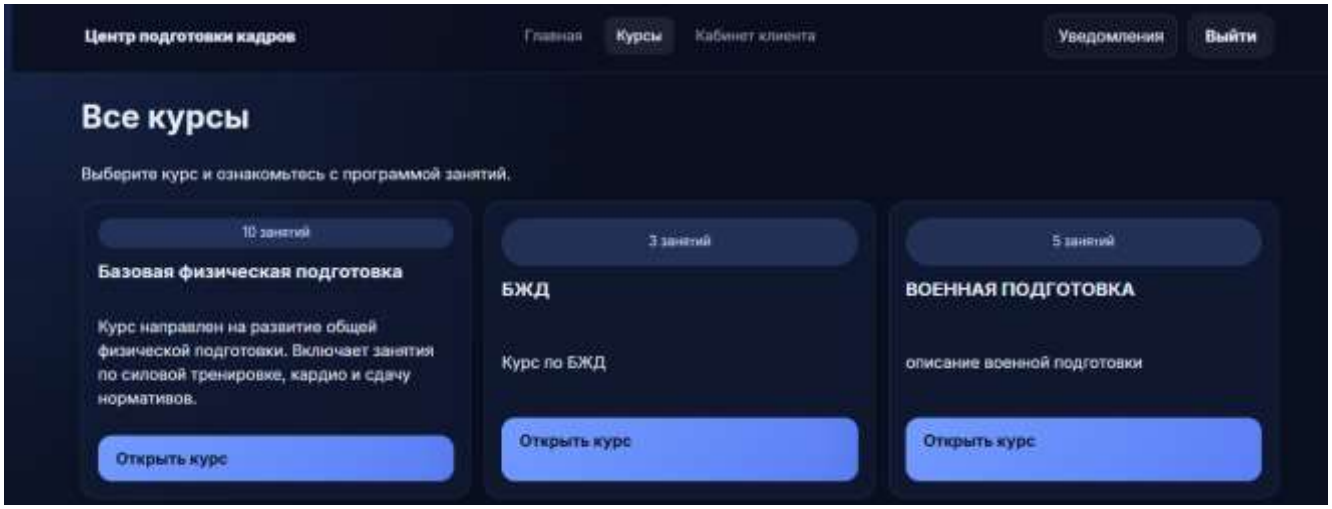


Рисунок 27 – Страница курсов

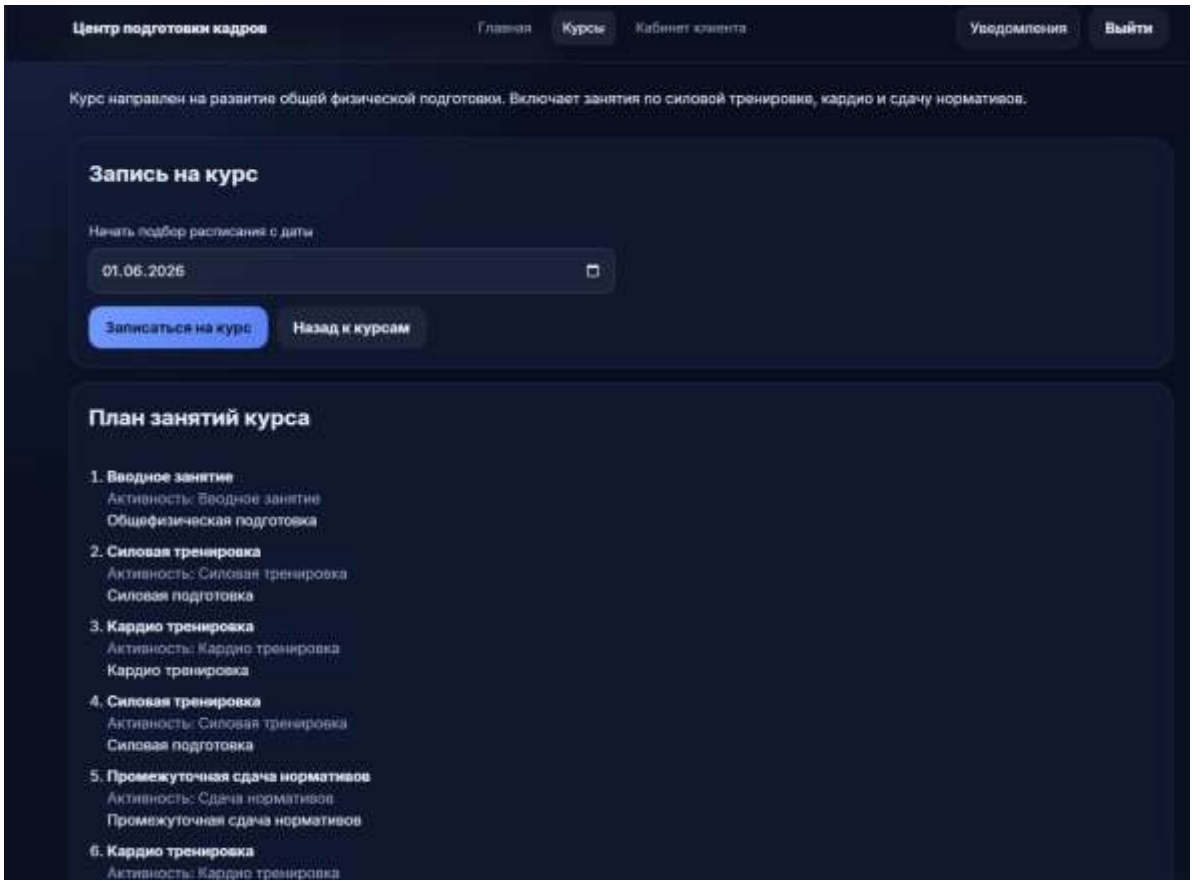


Рисунок 28 – Подробная информация о курсе

После записи персональное расписание формируется автоматически и становится доступным в разделе «Расписание на неделю». Этот раздел изображен на рисунке 29.

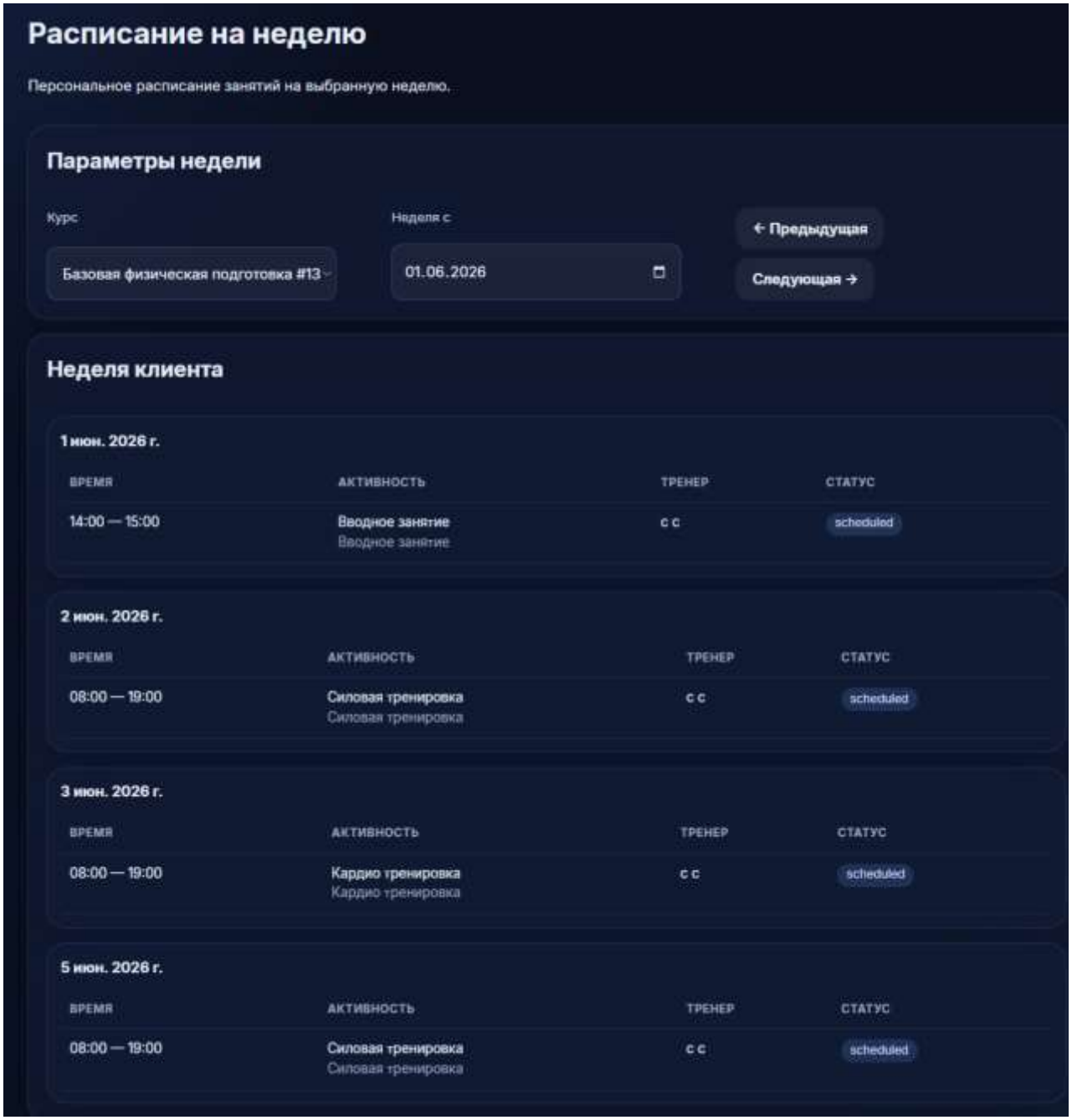


Рисунок 29 – Расписание недели клиента после записи на курс

После посещения занятий тренер отмечает, что клиент посетил тренировку и, если это была тренировка по сдаче нормативов, заполняет результаты занятия.

Также после записи на занятие клиент получает уведомление о том, что он успешно записан на курс, а тренер о том, что на его занятия записан новый клиент: Изображения уведомлений представлены на рисунках 30 и 31.

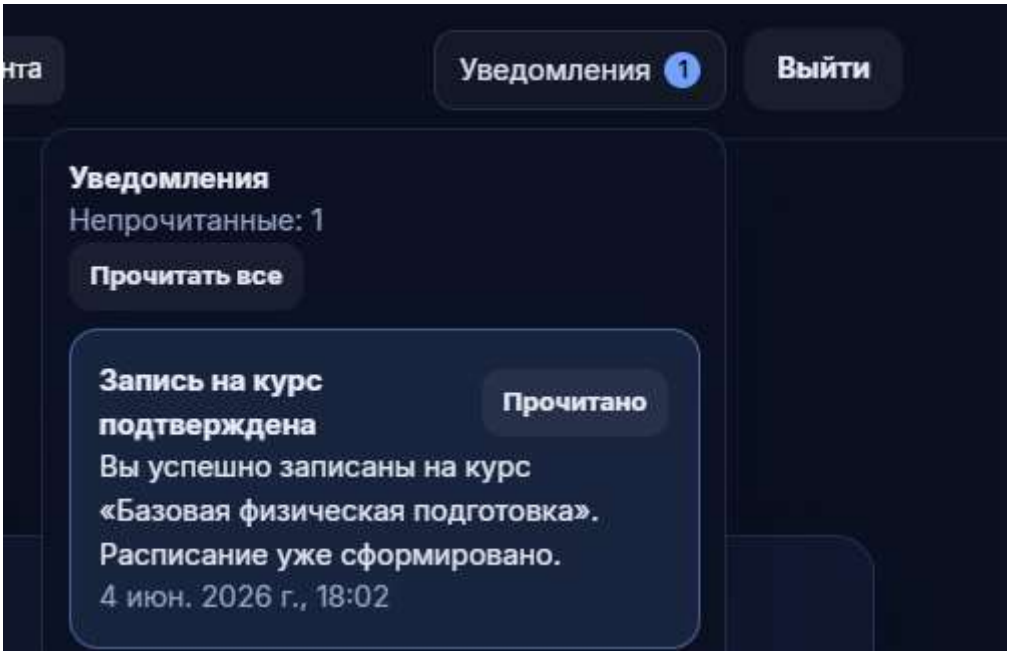


Рисунок 30 – Уведомление о записи на курс

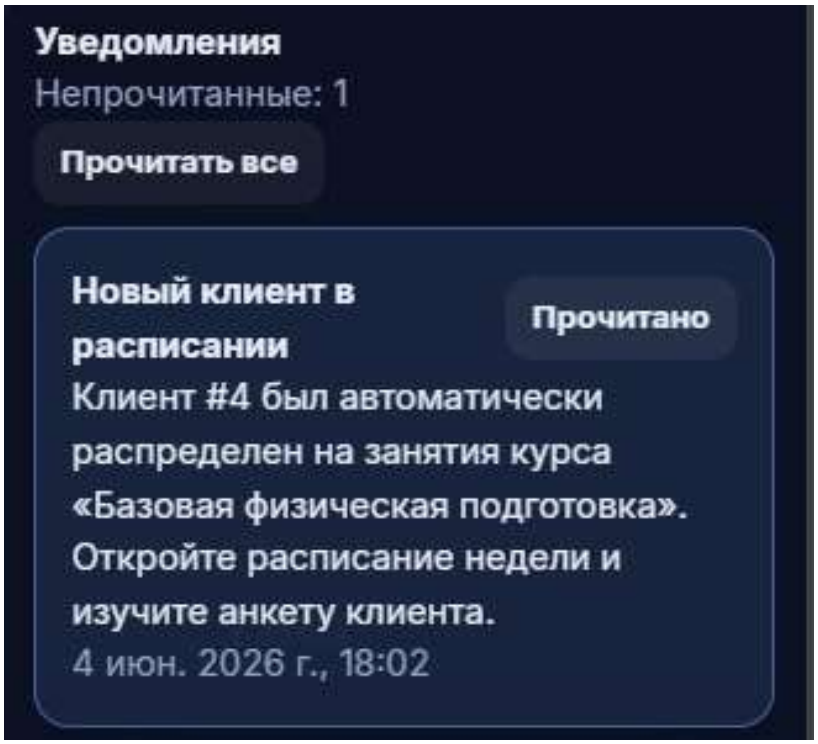


Рисунок 31 – Уведомление о записи на курс тренеру

Тренер после авторизации попадает в собственный кабинет, где доступны разделы расписания по дням с указанием клиентов в каждом слоте, карточки клиентов с анкетными данными, а также форма внесения результатов нормативов. Для отметки посещаемости тренер открывает нужный день, находит клиента в

списке и нажимает кнопку «Посетил» или «Пропуск». Кабинет тренера изображен на рисунке 32, а его расписание на рисунке 33.

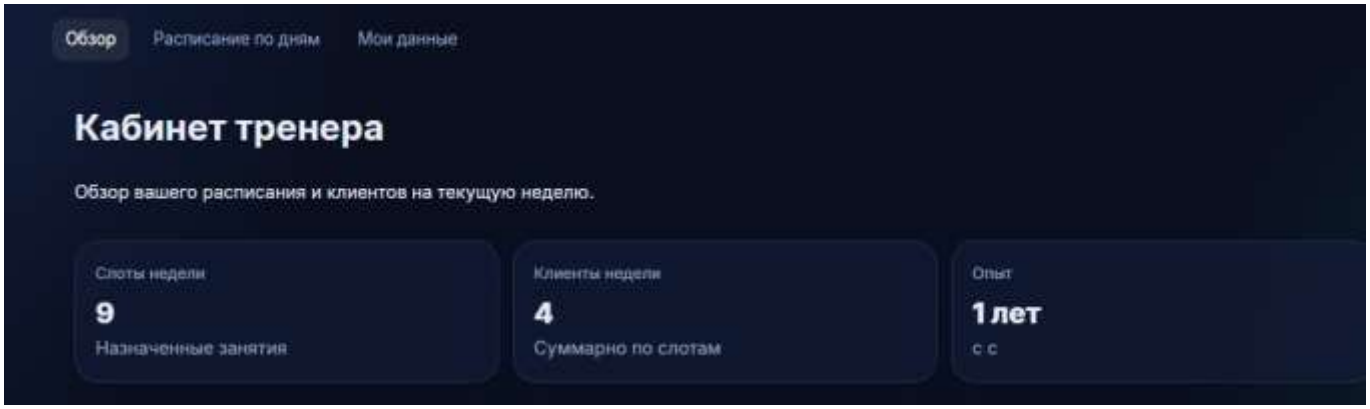


Рисунок 32 – Личный кабинет тренера

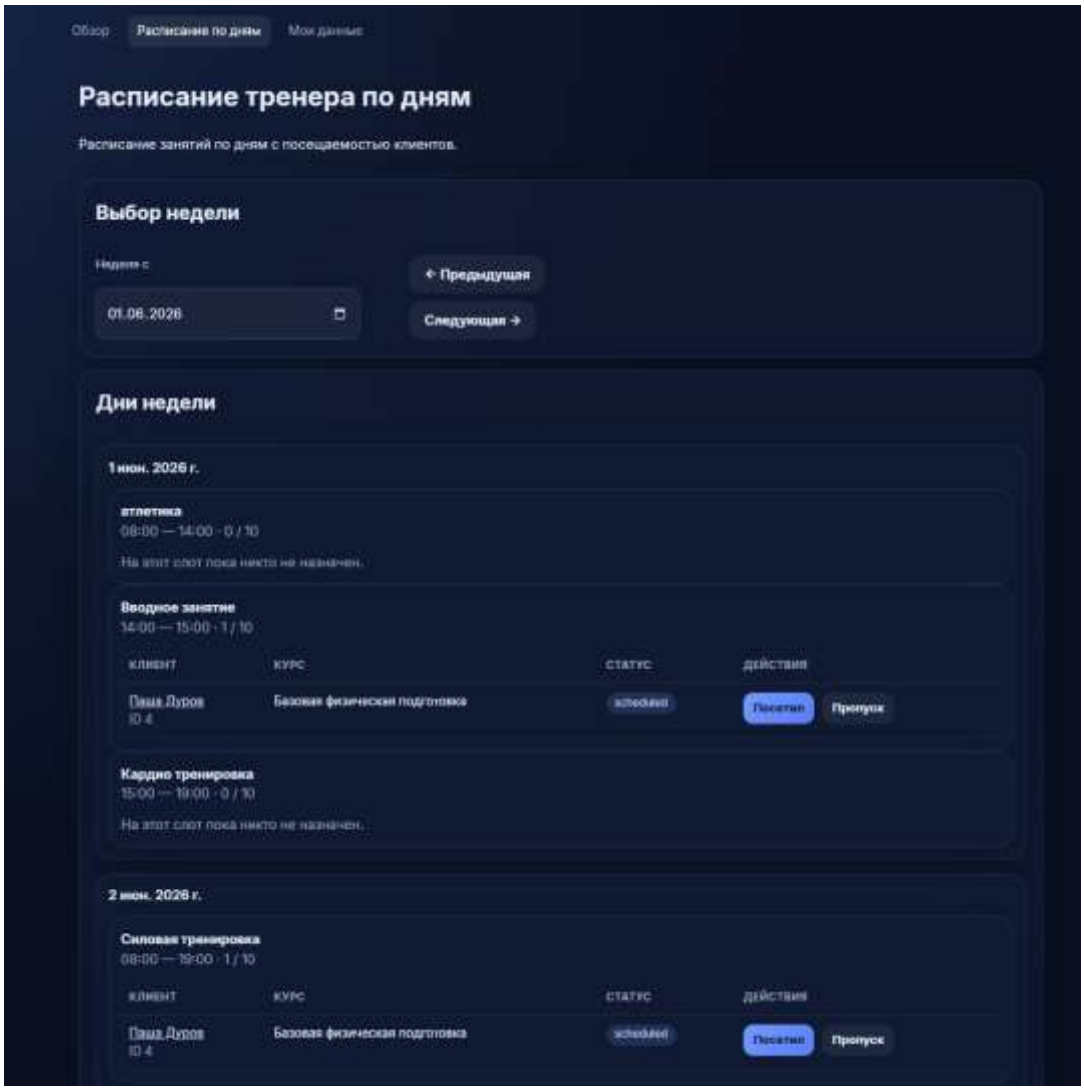


Рисунок 33 – Расписание тренера

Также тренер заполняет результаты сдачи норматива каждому клиенту: Форма заполнения нормативов представлена на рисунке 34.

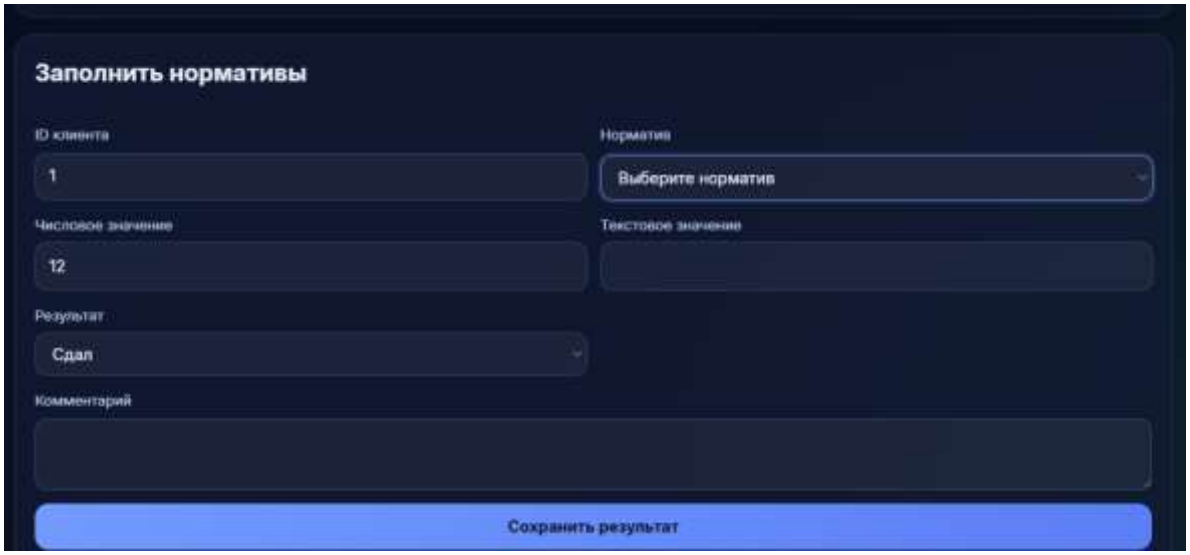


Рисунок 34 – Форма заполнения нормативов

Администратор после авторизации получает доступ к кабинету управления системой, включающему разделы: управление активностями, курсами, тренерами, нормативами, недельным расписанием и статьями. Для создания курса необходимо перейти в раздел «Курсы», заполнить название, описание и последовательно добавить шаги курса, указав для каждого шага тип активности. Недельное расписание формируется вручную в разделе «Расписание недели» - администратор добавляет слоты с указанием даты, времени, активности, тренера и вместимости группы. Готовое расписание можно продублировать на следующую неделю с помощью кнопки «Продублировать неделю». Создание расписания недели изображено на рисунке 35.

Для создания нового курса администратор указывает его название, описание, а также каждый шаг (занятие) курса. У каждого шага указывается его название, описание и выбирается активность.

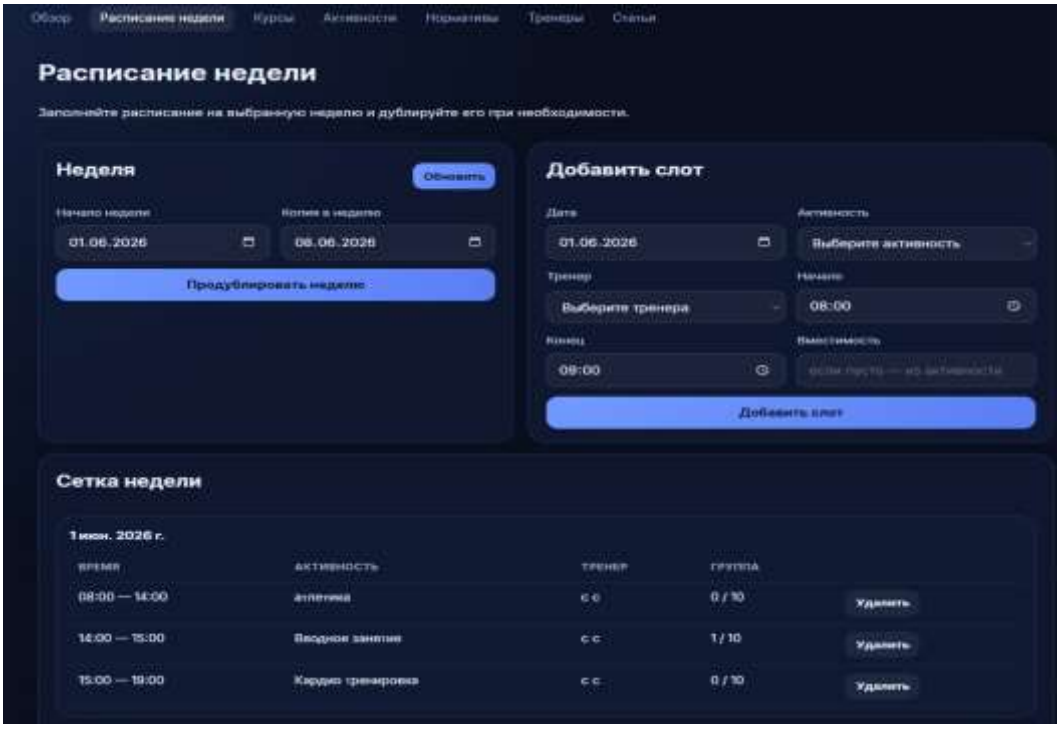


Рисунок 35 – Управление расписанием недели

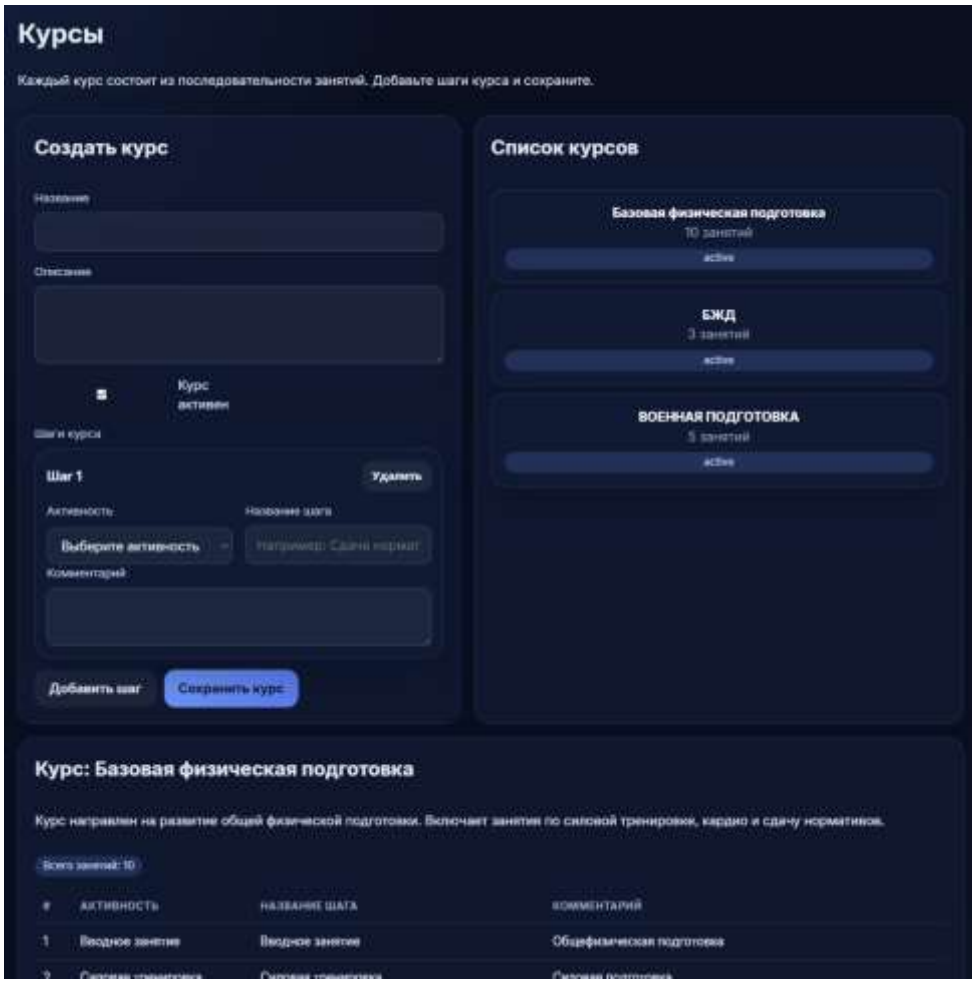


Рисунок 36 – Управление курсами

2 Охрана труда

Современное законодательство в области охраны труда при работе с электронно-вычислительными машинами (ЭВМ) выдвигает следующие требования:

– Помещение, в котором осуществляется работа с ПК, должно соответствовать следующим требованиям:

a) освещение должно соответствовать стандартам, а в случае отсутствия естественного освещения должны быть предоставлены расчеты, доказывающие безопасность работы без него;

b) оконные проемы должны иметь регулируемые устройства, такие как жалюзи или занавеси;

c) компьютеры не должны размещаться в местах, где есть риск попадания влаги.

d) рабочие места с компьютерами должны быть оборудованы защитным заземлением;

e) в помещениях с компьютерами следует проводить ежедневную влажную уборку и проветривание каждый час;

f) рекомендуемая температура воздуха в помещении с компьютерами составляет +19°C - +21°C;

g) рабочий стол должен быть спроектирован таким образом, чтобы обеспечить оптимальное расположение используемого оборудования в соответствии с выполняемой работой, разрешается использование различных типов рабочих столов, соответствующих современным стандартам эргономики;

h) в помещениях с компьютерами следует проводить ежедневную влажную уборку и проветривание каждый час.

Рабочие места с персональными компьютерами должны быть организованы следующим образом:

– при организации рабочих мест с компьютерами необходимо учитывать, что расстояние между рабочими столами, на которых установлены видеомониторы, должно быть не менее 2,0 метров в направлении между ними и не менее 1,2 метра между боковыми поверхностями видеомониторов;

– размещение компьютеров должно предусматривать расстояние не менее 1 метра от источников тепла. Следует избегать прямого попадания солнечного света на них;

– запрещается размещать рабочие места с компьютерами рядом с силовыми кабелями, высоковольтными трансформаторами и другим оборудованием, способным создавать помехи в работе компьютеров;

– при установке компьютеров следует избегать прохождения интерфейсных и электропитания кабелей по полу. В случае необходимости, их следует закрыть защитными коробками;

– экран видеомонитора должен находиться от глаз пользователя на расстоянии 600-700 мм, но не ближе 500 мм с учетом размеров алфавитно-цифровых знаков и символов;

– системный блок и монитор следует разместить так, чтобы расстояние от вентиляционных отверстий этих устройств до стены (или стенки мебели) составляло не менее 10 сантиметров;

– искусственное освещение в помещениях для эксплуатации компьютеров должно осуществляться системой общего равномерного освещения. Освещенность на поверхности рабочего стола в зоне размещения ПК должна быть 300-500 люкс. Освещение не должно создавать бликов на поверхности экрана. Освещенность поверхности экрана не должна быть более 300 люкс.

В случае работы на компьютере в течение 8-часовой смены следует предусматривать перерывы в соответствии с категорией работ:

– для работников I категории перерывы устанавливаются через 2 часа после начала смены и через 2 часа после обеденного перерыва продолжительностью 15 минут каждый;

– работники II категории должны иметь перерыв через 2 часа после начала смены и через 1,5 - 2,0 часа после обеденного перерыва продолжительностью 15 минут каждый или продолжительностью 10 минут через каждый час работы;

– для работников III категории рекомендуется устанавливать перерыв через 1,5 - 2,0 часа после начала смены и также через 1,5 - 2,0 часа после обеденного перерыва продолжительностью 20 минут каждый или продолжительностью 15 минут через каждый час работы.

При 12-часовой смене регламентированные перерывы должны быть предусмотрены в первые 8 часов работы так же, как и при 8-часовой смене. В течение последних 4 часов работы, независимо от категории и вида работ, необходимо предусматривать перерыв каждый час продолжительностью 15 минут.

3 Экономика

Разработка веб-приложения для центра подготовки кадров, выполненная в рамках выпускной квалификационной работы, представляет собой полноценный программный продукт уровня коммерческого проекта. В настоящем разделе приводится оценка стоимости такой разработки в случае её реализации силами профессионального разработчика на российском рынке труда. Подобный расчёт позволяет соотнести объём фактически проделанной работы с её рыночной стоимостью.

Общая длительность разработки составила пять календарных месяцев работы по графику пять рабочих дней в неделю при восьмичасовом рабочем дне, что соответствует приблизительно 105 рабочим дням. Ввиду того что проект охватывает как серверную, так и клиентскую части, а также включает настройку серверной инфраструктуры и контейнеризацию, для его реализации потребовался бы специалист уровня full-stack с компетенциями в области Python, FastAPI, SQLAlchemy, PostgreSQL, NATS, React, TypeScript, Docker и Nginx.

Согласно открытым данным сервисов hh.ru и «Хабр Карьера» по состоянию на 2026 год, средняя ежемесячная заработная плата full-stack разработчика уровня middle на российском рынке труда составляет около 280 000 рублей. С учётом полной занятости специалиста на протяжении всех пяти месяцев фонд оплаты труда составил бы 1 400 000 рублей. Дополнительно работодатель уплачивает страховые взносы в размере 30,2 процента от фонда оплаты труда, что добавляет к расчётной стоимости ещё около 423 000 рублей. Итого затраты на оплату труда с учётом страховых отчислений составляют порядка 1 823 000 рублей.

Помимо затрат на оплату труда, при реализации проекта необходимо учесть инфраструктурные и накладные расходы: лицензии на специализированное программное обеспечение и среды разработки (около 50 000 рублей), аренду виртуального сервера для размещения системы на год эксплуатации (около 18 000 рублей), регистрацию доменного имени в зоне .ru (около 1 000 рублей), накладные расходы на организацию рабочего места, оборудование и прочие

административные затраты (около 150 000 рублей), а также расходы на тестирование, документирование и подготовку к опытной эксплуатации (порядка 50 000 рублей). Итоговая совокупная стоимость разработки и первого года эксплуатации системы в коммерческой среде ориентировочно составляет 2 092 000 рублей.

В долгосрочной перспективе разработанная система способна достичь экономической окупаемости за счёт реализации по модели SaaS – предоставления доступа по подписке для центров спортивной и профессиональной подготовки. Целевая аудитория системы – спортивные секции, учебные центры и организации, занимающиеся подготовкой кадров, - испытывает устойчивую потребность в автоматизации процессов управления расписанием, учёта клиентов и контроля результатов обучения. Стоимость аналогичных SaaS-решений для управления учебными центрами на российском рынке составляет от 5 000 до 30 000 рублей в месяц на организацию. При базовом тарифе в 8 000 рублей в месяц и клиентской базе в 10 организаций ежемесячный доход составит 80 000 рублей, при росте клиентской базы до 30 организаций – 240 000 рублей в месяц. Дополнительный доход может быть получен за счёт расширенных тарифов с увеличенным числом пользователей и приоритетной технической поддержкой, а также разовых услуг по настройке системы под нужды конкретной организации.

При устойчивом росте клиентской базы проект способен выйти на самоокупаемость в течение 18–24 месяцев с момента коммерческого запуска. Использование исключительно открытого программного обеспечения – PostgreSQL, NATS, Nginx, Docker – исключает лицензионные отчисления и снижает операционные расходы при масштабировании системы.

Таким образом, с экономической точки зрения проект, разработанный в рамках выпускной квалификационной работы, с учётом заработной платы разработчика, страховых отчислений и инфраструктурных расходов, оценивается приблизительно в 2,0–2,1 миллиона рублей. Полученная оценка показывает, что выполненная работа по объёму и сложности соответствует полноценному коммерческому проекту. Наличие потенциала монетизации через модель подписки

для организаций обеспечивает возможность выхода проекта на самоокупаемость в среднесрочной перспективе, что делает разработку экономически обоснованной и практически востребованной на рынке автоматизации учебных центров.

Заключение

В ходе выполнения данной выпускной квалификационной работы было спроектировано и разработано веб-приложение для автоматизации управления центром подготовки кадров, представляющее собой полноценную клиент-серверную систему, доступную пользователю через современный веб-браузер с любого устройства.

В результате был проведён анализ предметной области управления учебными центрами и организации учебного процесса, рассмотрено текущее состояние рынка программных решений для автоматизации спортивных и образовательных организаций. Был выполнен подробный анализ существующих программных аналогов – рассмотрены наиболее распространённые на российском рынке системы: uni-gym.ru, утцзащитник.рф и alexfitness.ru. В ходе анализа выявлены их сильные и слабые стороны: большинство из них предоставляют базовые инструменты записи клиентов, однако не обеспечивают автоматического формирования персонального расписания при записи на курс, не разграничивают функциональность между ролями клиента и тренера в достаточной мере и не предоставляют тренеру инструментов для ведения посещаемости и нормативов непосредственно в системе. На основании проведённого анализа сформулированы требования к собственной разработке, призванной закрыть выявленную нишу комплексного инструмента управления центром подготовки с полноценными личными кабинетами для трёх категорий пользователей.

Перед началом разработки были выбраны программные средства и инструменты реализации. В качестве языка программирования серверной части выбран Python с фреймворком FastAPI, в качестве объектно-реляционного отображателя – SQLAlchemy, для версионирования схемы базы данных – Alembic, для аутентификации – PyJWT с асимметричными ключами по алгоритму RS256, для хеширования паролей – passlib. Для организации фоновых задач использованы брокер сообщений NATS с расширением JetStream и библиотека TaskIQ. Клиентская часть реализована на языке TypeScript с использованием библиотеки

React и инструмента сборки Vite. В качестве системы управления базой данных выбрана PostgreSQL. Развёртывание системы осуществляется с использованием Docker и Docker Compose, в качестве обратного прокси-сервера используется Nginx с SSL-сертификатом Let's Encrypt. Основной средой разработки выступила Visual Studio Code.

На основе приведённых технологий удалось спроектировать, разработать и протестировать веб-приложение со следующим функционалом:

– регистрация и авторизация пользователей с защищённым хранением паролей в виде криптографических хешей и аутентификацией на основе JWT-токенов с раздельным хранением access и refresh токенов;

– личный кабинет клиента с отображением активных курсов, персонального недельного расписания, медицинской анкеты, истории зачислений и динамики результатов по нормативам;

– личный кабинет тренера с расписанием занятий по дням, карточками клиентов с анкетными данными, инструментами отметки посещаемости и внесения результатов нормативов;

– административный кабинет с управлением справочниками активностей, курсов и нормативов, формированием недельного расписания с возможностью дублирования на следующую неделю, управлением тренерами и публикацией статей;

– автоматическое формирование персонального расписания клиента при записи на курс на основе недельной сетки занятий;

– система фоновых задач на базе брокера сообщений NATS для автоматической очистки истекших сессий пользователей и рассылки уведомлений;

– разграничение прав доступа для трёх ролей пользователей: клиента, тренера и администратора – с защитой маршрутов на уровне как серверной, так и клиентской части.

Корректность работы реализованных функций подтверждена двумя взаимодополняющими подходами: автоматическими тестами на фреймворке pytest,

а также ручным тестированием наиболее распространённых пользовательских сценариев и обработки исключительных ситуаций.

Разработанная в рамках данной выпускной квалификационной работы система на текущий момент содержит весь необходимый базовый функционал для начала комфортной работы. Однако, как и большинство других программных продуктов, она допускает дальнейшее развитие в плане реализации новых функций и улучшения работы действующих. В качестве перспективных направлений доработки могут рассматриваться: реализация мобильного приложения для iOS и Android, добавление модуля онлайн-оплаты курсов, расширение инструментов аналитики для администратора с формированием отчётов по посещаемости и результатам клиентов, а также интеграция с внешними корпоративными системами учебных организаций.

Созданное веб-приложение для центра подготовки кадров, призванное упростить и автоматизировать процессы управления курсами, расписанием и взаимодействием между участниками учебного процесса, полностью готово к опытной эксплуатации. Наличие трёх полноценных личных кабинетов, автоматического формирования персонального расписания, системы фоновых задач и контейнеризированного развёртывания закрывают ту нишу, в которой существующие аналоги представлены лишь частично, что делает разработанный программный продукт востребованным как для небольших спортивных секций, так и для крупных центров профессиональной подготовки кадров.

Список использованных источников

1. Матухно Е.В. Профессионально-прикладная физическая подготовка. - Комсомольск-на-Амуре: ФГБОУ ВПО «КнАГТУ», 2013. – 97 с.
2. Теория и методика физического воспитания и спорта: Учеб. пособие для студ. высш. учеб.заведений. -2-е изд. М.: Издательский центр «Академия», 2003. – 481с.
3. Основные направления цифровой трансформации физической культуры и спорта в России – известия ТулГУ. Физическая культура. Спорт. 2021. Вып.9. – 106 с.
4. Доусон, М. Программируем на Python / М. Доусон. – Санкт-Петербург: Питер, 2014. – 416 с.
5. Рогов Е. В. PostgreSQL 16 изнутри. – М.: ДМК Пресс, 2024. – 664 с.
6. SQLAlchemy 2.0 Documentation. Официальный сайт документации. [Электронный ресурс]. – Режим доступа: https://docs.sqlalchemy.org/en/20/ (дата обращения 21.03.2026)
7. FastAPI Documentation. Официальная документация фреймворка. [Электронный ресурс]. – Режим доступа: https://fastapi.tiangolo.com/ru/ (дата обращения 03.04.2026)
8. Официальная документация Taskiq. [Электронный ресурс] / Режим доступа: https://taskiq-python.github.io/guide/getting-started.html (дата обращения 17.04.2026).
9. NATS Documentation. Официальная документация NATS. [Электронный ресурс]. – Режим доступа: https://docs.nats.io (дата обращения 25.04.2026)
10. Dishka Documentation. Официальная документация Dishka. [Электронный ресурс]. – Режим доступа: https://dishka.readthedocs.io/en/stable/index.html (дата обращения 01.04.2026)
11. Стив ван М., Таненбаум Э.С. Распределенные системы / пер. с англ. В.А. Яроцкого. – М.: ДМК Пресс, 2021. – 584 с.

12. Роберт Мартин. Чистая архитектура. Искусство разработки программного обеспечения – СПб: Питер, 2023. – 352 с.

13. Alembic Documentation. Официальная документация Alembic. [Электронный ресурс]. – Режим доступа: https://alembic.sqlalchemy.org/en/latest/index.html (дата обращения 29.03.2026)

14. JavaScript Documentation. Официальный сайт библиотеки. [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата обращения 21.04.2026)

15. TypeScript Handbook. Официальная документация языка программирования. [Электронный ресурс]. – Режим доступа: https://www.typescriptlang.org/docs/ (дата обращения 22.05.2026)

16. pytest Documentation. Документация фреймворка автоматического тестирования. [Электронный ресурс]. – Режим доступа: https://docs.pytest.org/en/stable/ (дата обращения 11.04.2026).

17. Docker Documentation. Официальный сайт библиотеки. [Электронный ресурс]. – Режим доступа: https://docs.docker.com (дата обращения 08.05.2026)

18. Git Documentation. Документация по системе контроля версий. [Электронный ресурс]. – Режим доступа https://git-scm.com/docs/git (дата обращения 20.05.2026)

Приложение А

(обязательное)

WEB-приложение для центра подготовки кадров

UML-диаграммы

Приложение А.1 – Диаграмма вариантов использования клиента

ПГУ 1.090501.13.002 Д3

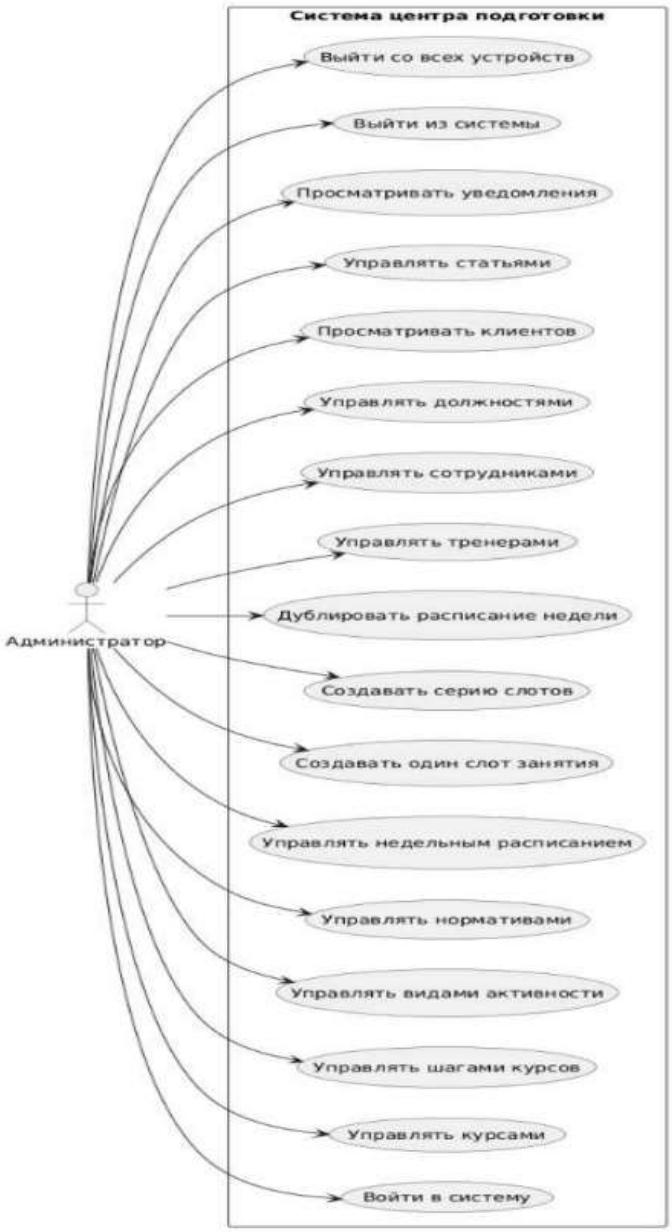
ФВТ, каф. ВТ
гр. 21ВОЭ1

Приложение А.2 – Диаграмма вариантов использования тренера

ПГУ 1.090501.13.003 Д3

ФВТ, каф. ВТ
гр. 21ВОЭ1

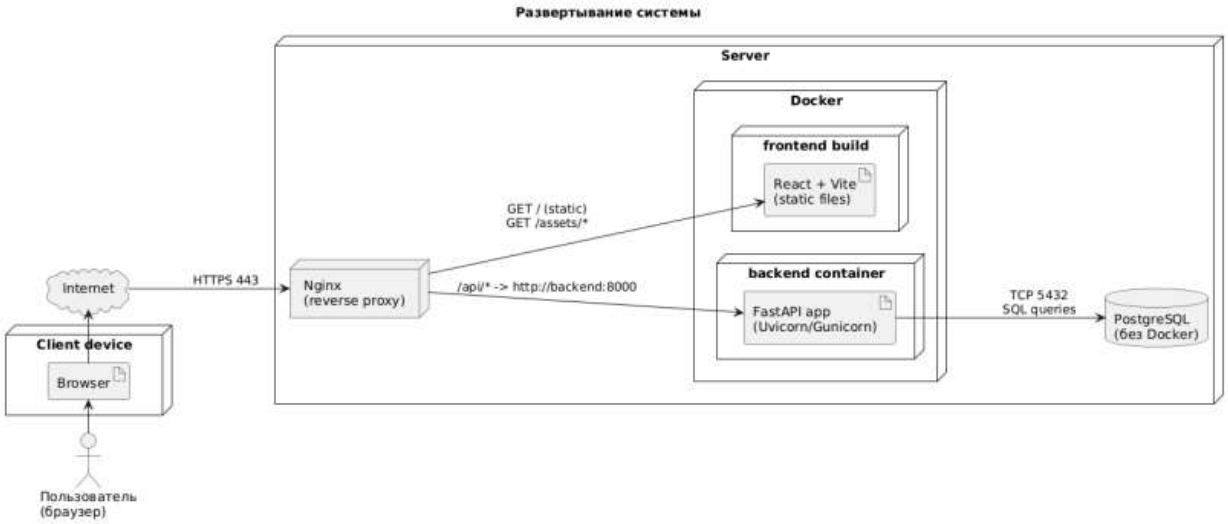
Приложение А.3 – Диаграмма вариантов использования администратора



ПГУ 1.090501.13.004 Д18

ФВТ, каф. ВТ
гр. 21ВОЭ1

Приложение А.4 – Диаграмма развертывания системы



ПГУ 1.090501.13.005 Д19

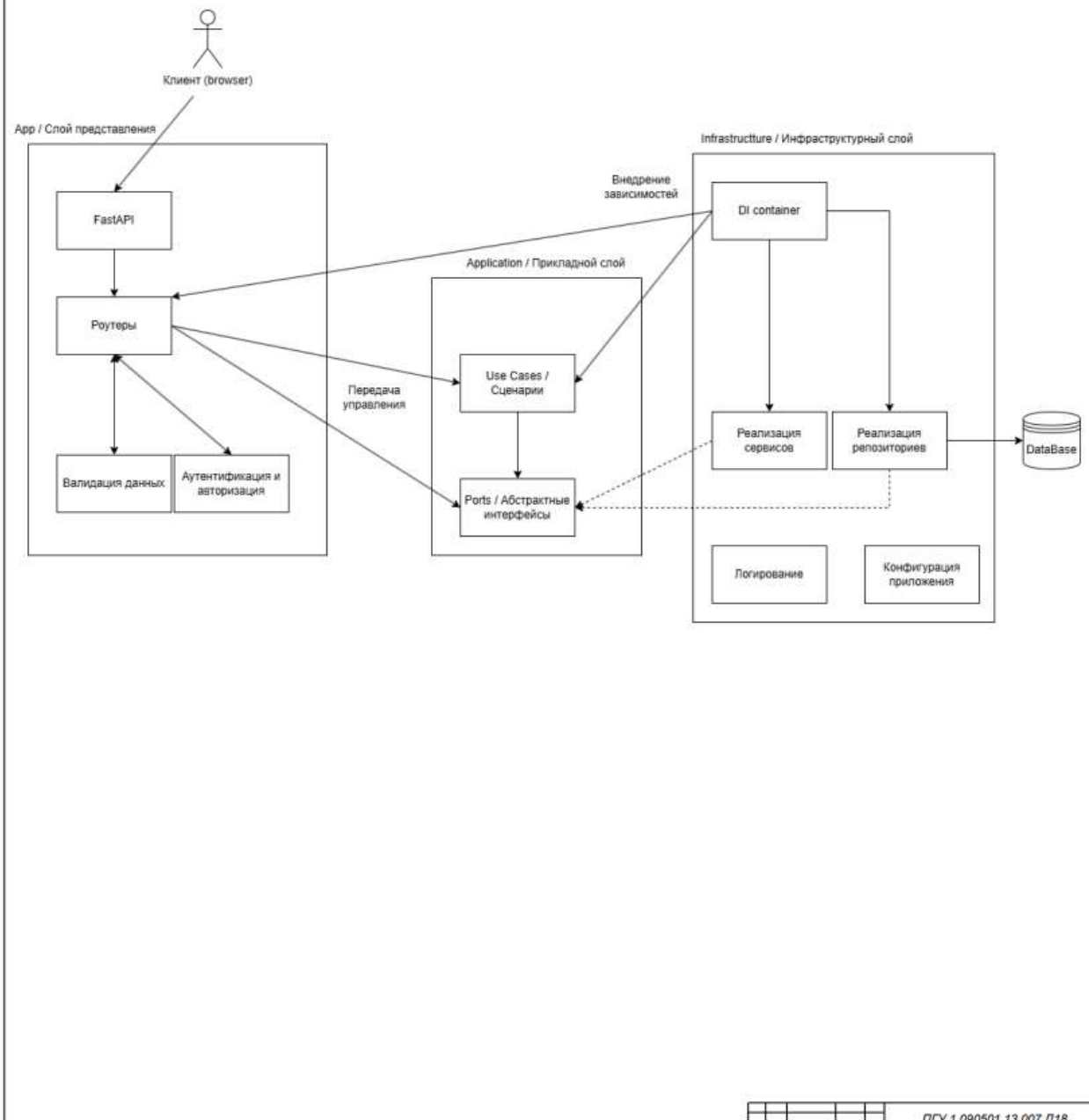
ФВТ, каф. ВТ
гр. 21ВОЭ1

Приложение А.5 – Диаграмма последовательности для основного сценария работы приложения

ПГУ 1.090501.13.006 Д14

ФВТ, каф. ВТ
гр. 21ВОЭ1

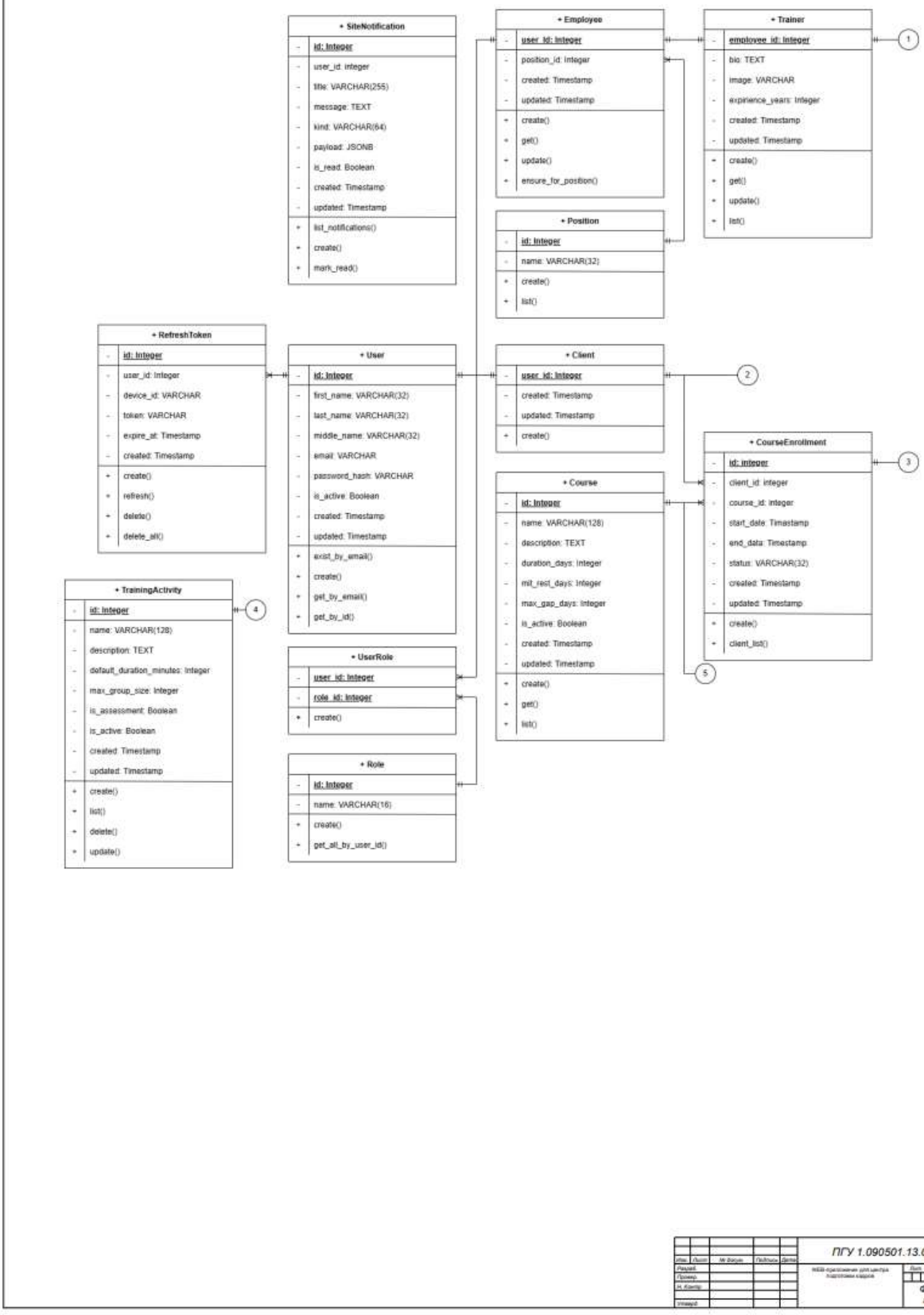
Приложение А.6 – Компонентная диаграмма системы



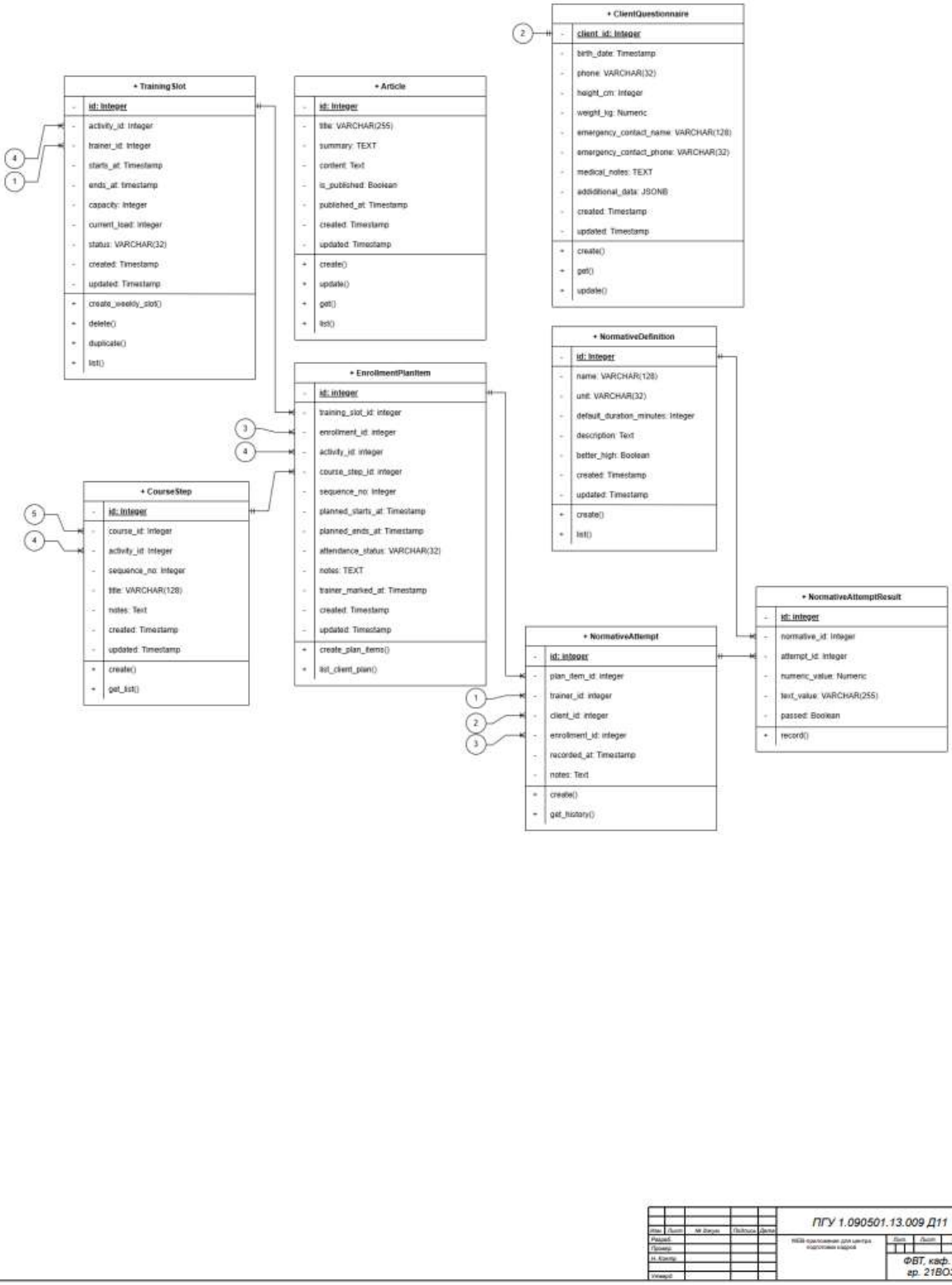
ПГУ 1.090501.13.007 Д18

ФВТ, каф. ВТ
гр. 21ВОЭ1

Приложение А.7 – Диаграмма классов серверной части (часть 1)



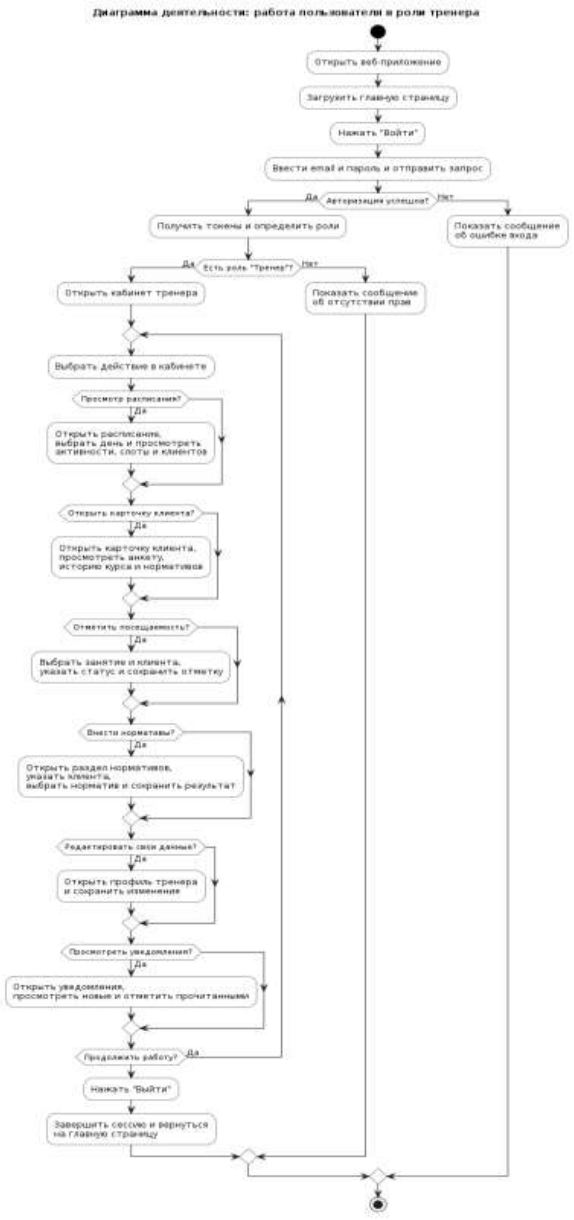
Приложение А.8 – Диаграмма классов серверной части (часть 2)



ПГУ 1.090501.13.009 Д11

ФВТ, каф. ВТ
гр. 21ВОЭ1

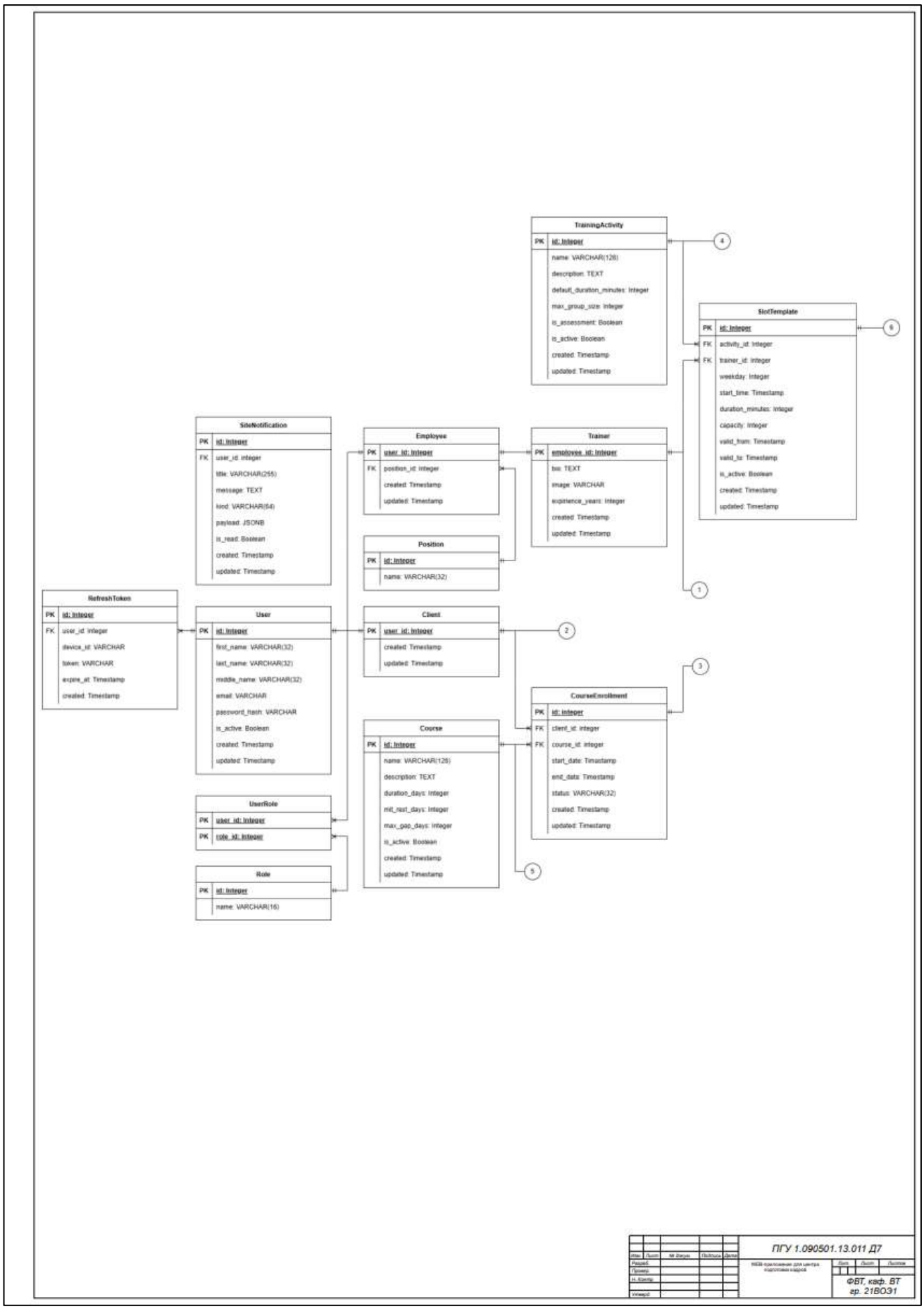
Приложение А.9 – Диаграмма деятельности пользователя системы



ПГУ 1.090501.13.010 Д17

ФВТ, каф. ВТ
гр. 21ВОЭ1

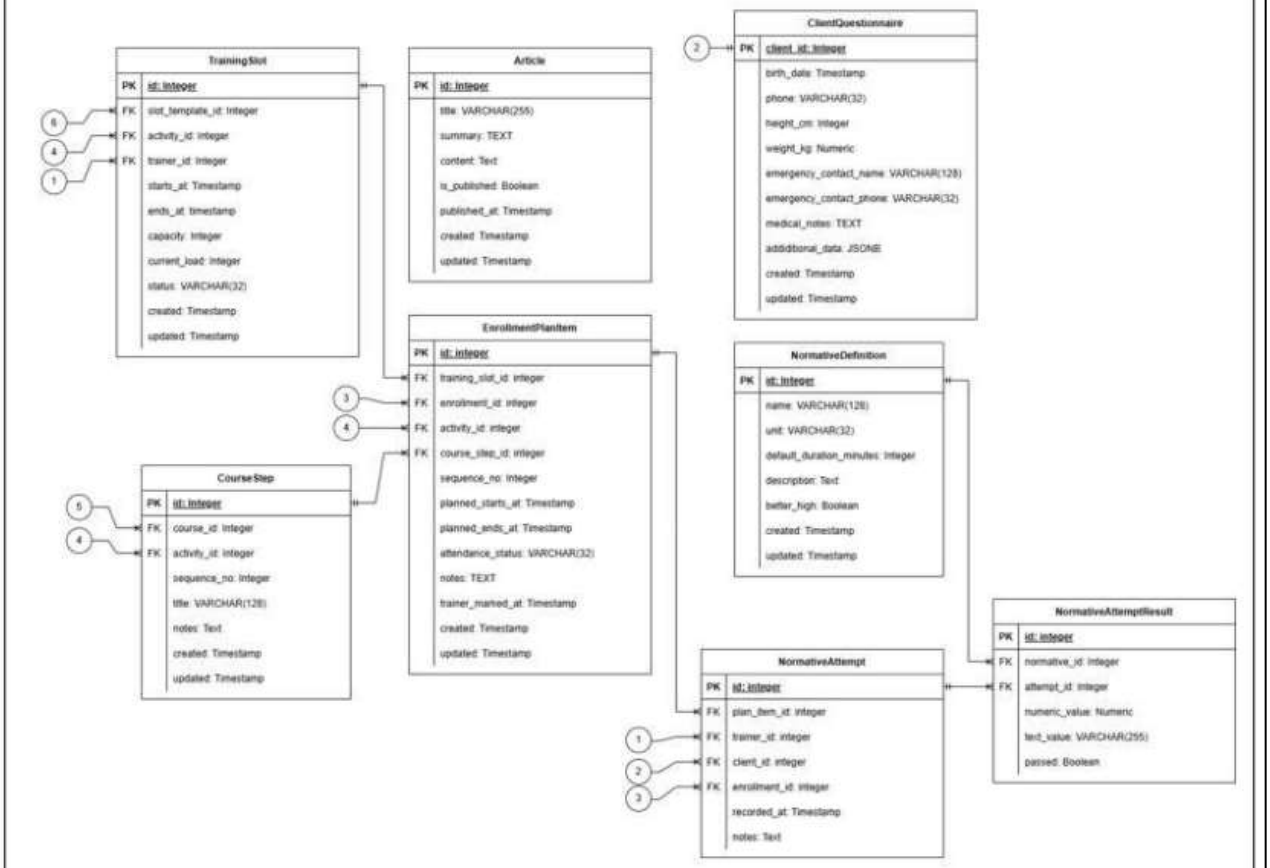
Приложение А.10 – Физическая модель базы данных (часть 1)



ПГУ 1.090501.13.011 Д7

ФВТ, каф. ВТ
гр. 21ВОЭ1

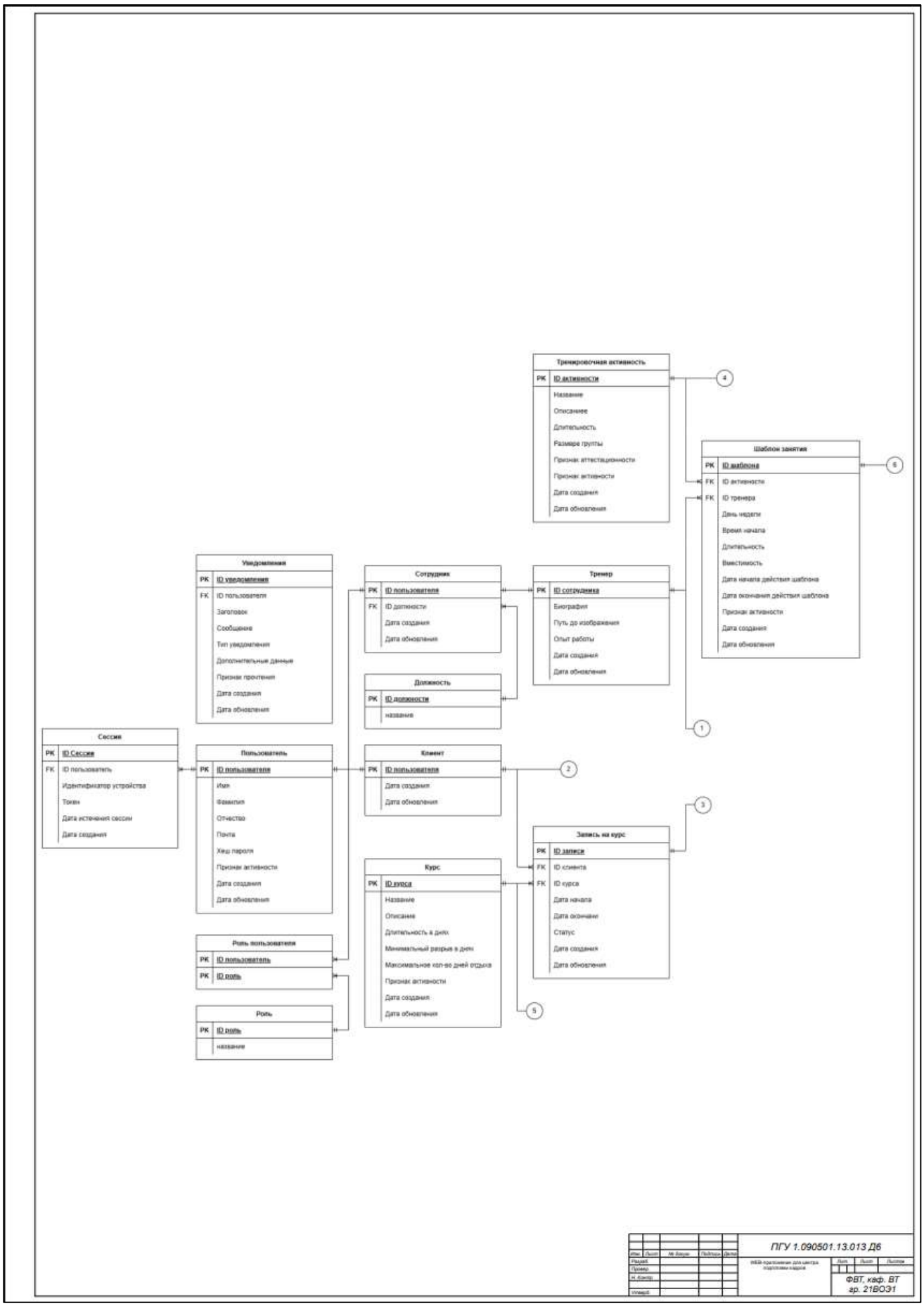
Приложение А.11 – Физическая модель базы данных (часть 2)



ПГУ 1.090501.13.012 Д7

ФВТ, каф. ВТ
гр. 21ВОЭ1

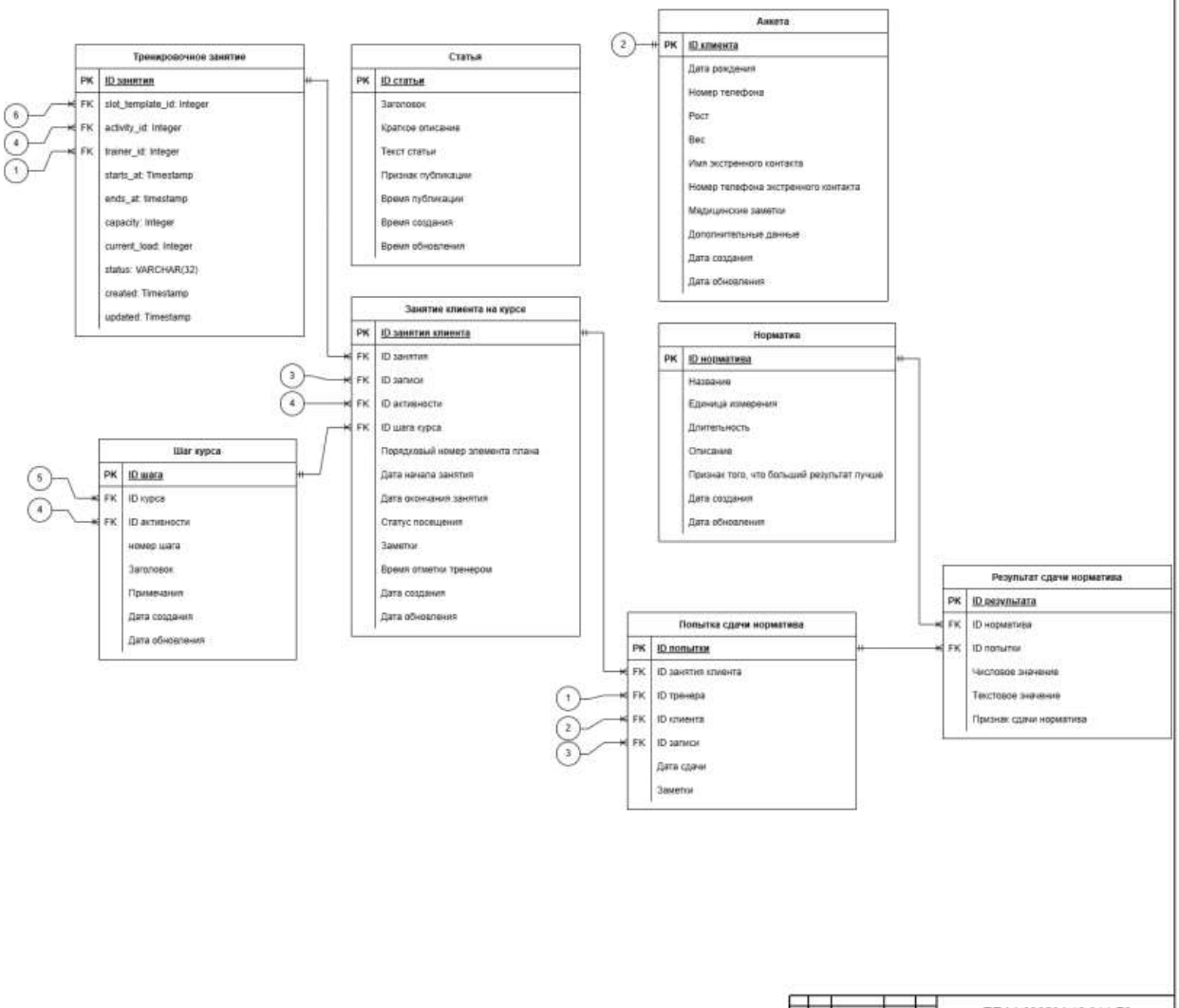
Приложение А.12 – Логическая схема базы данных (часть 1)



ПГУ 1.090501.13.013 Д6

ФВТ, каф. ВТ
гр. 21ВОЭ1

Приложение А.13 – Логическая схема базы данных (часть 2)



Приложение Б

(обязательное)

WEB-приложение для центра подготовки кадров

Листинги программы

Серверная часть приложения

Файл /api/v1/endpoints/client.py:

```
from datetime import date, datetime, time, timedelta
from dishka import FromDishka
from dishka.integrations.fastapi import DishkaRoute
from fastapi import APIRouter, Depends, HTTPException
from starlette import status
from src.api.security.role_filter import get_current_client
from src.application.core.exceptions.training_center import
EnrollmentAlreadyExists, PlanningFailed
from src.application.ports.services.enrollment_tc import
EnrollmentServicePort
from src.application.schemas.client import ClientProfileRead
from src.application.schemas.enrollment import EnrollCourseRequest,
EnrollmentRead, NormativeHistoryPoint, PlanItemRead
from src.application.schemas.questionnaire import QuestionnaireRead,
QuestionnaireUpsert
from src.application.schemas.user import UserBaseSchema
from src.application.usecases.training_center import
EnrollOnCourseUseCase, UpsertQuestionnaireUseCase

client_router = APIRouter(prefix="/client", tags=["client"],
route_class=DishkaRoute)

@client_router.get("/profile", response_model=ClientProfileRead)
async def get_client_profile(
    enrollment_service: FromDishka[EnrollmentServicePort],
    user: UserBaseSchema = Depends(get_current_client),
):
    profile = await enrollment_service.get_client_profile(user.id)
    if profile is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Client not found")
    return profile


@client_router.get("/questionnaire", response_model=QuestionnaireRead)
async def get_questionnaire(
    enrollment_service: FromDishka[EnrollmentServicePort],
    user: UserBaseSchema = Depends(get_current_client),
):
    questionnaire = await
enrollment_service.get_questionnaire(user.id)
    if questionnaire is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Questionnaire not filled yet")
    return questionnaire


@client_router.put("/questionnaire", response_model=QuestionnaireRead)
```

```
async def upsert_questionnaire(
    payload: QuestionnaireUpsert,
    usecase: FromDishka[UpsertQuestionnaireUseCase],
    user: UserBaseSchema = Depends(get_current_client),
):
    return await usecase(user.id, payload)


@client_router.post("/enrollments", response_model=EnrollmentRead)
async def enroll_on_course(
    payload: EnrollCourseRequest,
    usecase: FromDishka[EnrollOnCourseUseCase],
    user: UserBaseSchema = Depends(get_current_client),
):

    try:
        return await usecase(user.id, payload.course_id,
payload.start_date or date.today())
    except EnrollmentAlreadyExists:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT,
detail="Client already has active enrollment for this course")
    except PlanningFailed as exc:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT,
detail=str(exc))


@client_router.get("/enrollments",
response_model=list[EnrollmentRead])
async def list_client_enrollments(
    enrollment_service: FromDishka[EnrollmentServicePort],
    user: UserBaseSchema = Depends(get_current_client),
):
    return await enrollment_service.list_client_enrollments(user.id)


@client_router.get("/plan", response_model=list[PlanItemRead])
async def get_client_plan(
    enrollment_service: FromDishka[EnrollmentServicePort],
    enrollment_id: int | None = None,
    user: UserBaseSchema = Depends(get_current_client),
):
    return await enrollment_service.list_client_plan(user.id,
enrollment_id=enrollment_id)


@client_router.get("/week-plan", response_model=list[PlanItemRead])
async def get_client_week_plan(
    week_start: date,
    enrollment_service: FromDishka[EnrollmentServicePort],
    enrollment_id: int | None = None,
    user: UserBaseSchema = Depends(get_current_client),
):
    start_at = datetime.combine(week_start, time.min)
```

```
    end_at = start_at + timedelta(days=7)
    items = await enrollment_service.list_client_plan(user.id,
enrollment_id=enrollment_id)
    return [item for item in items if start_at <=
item.planned_starts_at < end_at]


@client_router.get("/normatives",
response_model=list[NormativeHistoryPoint])
async def get_normative_history(
    enrollment_service: FromDishka[EnrollmentServicePort],
    course_id: int | None = None,
    user: UserBaseSchema = Depends(get_current_client),
):
    return await enrollment_service.get_normative_history(user.id,
course_id=course_id)
```

Файл /api/v1/endpoints/trainer.py

```
@trainer_router.get("/profile", response_model=TrainerRead)
async def get_profile(
    trainer_service: FromDishka[TrainerServicePort],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    trainer = await trainer_service.get(user.id)
    if trainer is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Trainer not found")
    return trainer

@trainer_router.patch("/profile",
status_code=status.HTTP_204_NO_CONTENT)
async def update_profile(
    payload: TrainerUpdate,
    usecase: FromDishka[UpdateTrainerProfileUseCase],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    try:
        await usecase(user.id, payload)
    except TrainerNotFound:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Trainer not found")


@trainer_router.get("/slots", response_model=list[TrainingSlotRead])
async def list_own_slots(
    start_at: datetime,
    end_at: datetime,
    schedule_service: FromDishka[ScheduleServicePort],
    user: UserBaseSchema = Depends(get_current_trainer),
):
```

```
    return await schedule_service.list_slots(start_at, end_at,
trainer_id=user.id)
@trainer_router.get("/week-schedule",
response_model=list[TrainerWeekSlotRead])
async def get_week_schedule(
    week_start: date,
    schedule_service: FromDishka[ScheduleServicePort],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    return await schedule_service.list_trainer_week_slots(user.id,
week_start)


@trainer_router.get("/clients/{client_id}",
response_model=ClientProfileRead)
async def get_client_profile_for_trainer(
    client_id: int,
    enrollment_service: FromDishka[EnrollmentServicePort],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    try:
        profile = await
enrollment_service.get_client_profile_for_trainer(user.id, client_id)
    except ForbiddenTrainerAccess:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="No access to this client")
    if profile is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Client not found")
    return profile


@trainer_router.patch("/plan-items/{plan_item_id}/attendance",
status_code=status.HTTP_204_NO_CONTENT)
async def mark_attendance(
    plan_item_id: int,
    payload: AttendanceMarkRequest,
    usecase: FromDishka[MarkAttendanceUseCase],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    try:
        await usecase(plan_item_id, user.id, payload)
    except PlanItemNotFound:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Plan item not found")
    except ForbiddenTrainerAccess:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
detail="You can mark only your own slot")


@trainer_router.get("/normatives/catalog",
response_model=list[NormativeRead])
async def get_normatives_catalog(
```

```
    catalog_service: FromDishka[CatalogServicePort],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    return await catalog_service.list_normatives()


@trainer_router.post("/normatives/results",
status_code=status.HTTP_204_NO_CONTENT)
async def record_normatives(
    payload: RecordNormativeResultsRequest,
    usecase: FromDishka[RecordNormativeResultsUseCase],
    user: UserBaseSchema = Depends(get_current_trainer),
):
    try:
        await usecase(user.id, payload)
    except EnrollmentNotFound:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Active enrollment for client not found")
    except NormativeNotFound:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Normative not found")
```

Файл /api/core/setup.py

```
from contextlib import asynccontextmanager
from fastapi import FastAPI, APIRouter
from fastapi.middleware.cors import CORSMiddleware
from starlette.requests import Request
from starlette.responses import JSONResponse
from starlette.staticfiles import StaticFiles
from src.api.v1.endpoints.admin_tc import admin_router
from src.api.v1.endpoints.auth import auth_router
from src.api.v1.endpoints.content import admin_content_router,
notifications_router, public_content_router
from src.api.v1.endpoints.client import client_router
from src.api.v1.endpoints.courses import courses_router
from src.api.v1.endpoints.trainer import trainer_router
from src.infrastructure.core.logger import get_logger
from src.infrastructure.core.settings import settings
from src.infrastructure.di.setup import init_di
from src.infrastructure.taskiq.broker import source, broker, scheduler
log = get_logger(__name__)

async def unhandled_exception_handler(request: Request, exc:
Exception):
    log.error(
        "Unhandled exception",
        exc_info=exc,
        extra={"path": request.url.path, "method": request.method},
    )
    return JSONResponse(status_code=500, content={"detail": "Internal
```

```
server error"})

@asynccontextmanager
async def lifespan(app: FastAPI):
    await source.startup()
    if not broker.is_worker_process:
        await scheduler.startup()

    yield

    await source.shutdown()
    if not broker.is_worker_process:
        await scheduler.shutdown()


def register_routes(app: FastAPI):
    api_router = APIRouter(prefix="/api")
    api_router.include_router(auth_router)
    api_router.include_router(courses_router)
    api_router.include_router(public_content_router)
    api_router.include_router(notifications_router)
    api_router.include_router(client_router)
    api_router.include_router(trainer_router)
    api_router.include_router(admin_router)
    api_router.include_router(admin_content_router)
    app.include_router(api_router)


def register_exception_handlers(app: FastAPI):
    app.add_exception_handler(Exception, unhandled_exception_handler)


def create_fastapi_app() -> FastAPI:
    app = FastAPI(
        title="Training Center API",
        version="1.0.0",
        docs_url="/docs",
        redoc_url="/redoc",
        openapi_url="/openapi.json",
        lifespan=lifespan,
    )
    init_di(app)
    register_routes(app)
    register_exception_handlers(app)
    app.add_middleware(
        CORSMiddleware,
        allow_origins=[
            "http://localhost:5173",
            "http://127.0.0.1:5173",
        ],
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
```

```
    )
    app.mount("/media", StaticFiles(directory=settings.static_path),
name="media")
    return app
```

Файл /application/ports/services/user.py

```
class UserServicePort(Protocol):
    async def create(self, data: SignUpRequest) -> UserSchema: ...
    async def exist_by_email(self, email: str) -> bool: ...
    async def get_by_email(self, email: str) -> UserSchema | None: ...
    async def get_by_id(self, user_id: int) -> UserSchema | None: ...
    async def get_full_info_by_id(self, user_id: int) ->
UserFullInfoSchema | None: ...
```

Файл /application/ports/services/catalog_tc.py

```
class CatalogServicePort(Protocol):
    async def create_activity(self, data: ActivityCreate) ->
ActivityRead: ...
    async def list_activities(self) -> list[ActivityRead]: ...
    async def create_normative(self, data: NormativeCreate) ->
NormativeRead: ...
    async def list_normatives(self) -> list[NormativeRead]: ...
    async def create_course(self, data: CourseCreate) -> CourseRead:
...
    async def get_course(self, course_id: int) -> CourseRead | None:
...
    async def list_courses(self, active_only: bool = True) ->
list[CourseShortRead]: ...
```

Файл /application/ports/repositories/user.py

```
class UserRepositoryPort(Protocol):
    async def create(self, data: SignUpRequest) -> UserSchema: ...
    async def exist_by_email(self, email: str) -> bool: ...
    async def get_by_email(self, email: str) -> UserSchema | None: ...
    async def get_by_id(self, user_id: int) -> UserSchema | None: ...
    async def get_full_info_by_id(self, user_id: int) ->
UserFullInfoSchema | None: ...
```

Файл /application/schemas/user.py

```
class UserBaseSchema(BaseModel):
    id: int
    roles: list[str | None] = Field(default_factory=list)
```

```
class UserSchema(UserBaseSchema):
    email: EmailStr
    password_hash: str

class UserFullInfoSchema(UserSchema):
    first_name: str
    last_name: str
    middle_name: str | None
```

Файл /application/schemas/auth.py

```
class SignUpRequest(BaseModel):
    email: EmailStr
    password: str
    first_name: str
    last_name: str
    middle_name: str | None

class LoginRequest(BaseModel):
    email: EmailStr
    password: str


class SuccessSignUpSchema(BaseModel):
    access_token: str
    refresh_token: str
    token_type: str = "bearer"
    user: UserSchema

class SuccessLoginSchema(BaseModel):
    access_token: str
    refresh_token: str
    token_type: str = "bearer"
    user: UserSchema

class SuccessRefreshTokensSchema(BaseModel):
    access_token: str
    refresh_token: str


class AuthResponse(BaseModel):
    access_token: str
    token_type: str = "bearer"
```

Файл /application/usecases/auth/signin

```
class LoginUseCase:
    def __init__(
        self,
        user_service: UserServicePort,
        user_session_service: UserSessionServicePort,
        jwt_processor: JWTProcessorPort,
        password_processor: PasswordProcessorPort,
        session: AsyncSession,
        settings: Settings
    ):
        self.user_service = user_service
        self.user_session_service = user_session_service
        self.jwt_processor = jwt_processor
        self.password_processor = password_processor
        self.settings = settings

        self.__session = session

    async def __call__(
        self,
        email: str,
        password: str,
        client_info: RequestClient
    ) -> SuccessLoginSchema:
        try:
            async with self.__session.begin():
                user = await
self.user_service.get_by_email(email=email)
                if not user or not
self.password_processor.verify_password(
                    password=password,
hashed_password=user.password_hash
                ):
                    raise InvalidLoginData(f"Incorrect email or
password")

                access_token =
self.jwt_processor.generate_access_token(
                    user_data=UserBaseSchema(
                        id=user.id,
                        roles=user.roles
                    ),
                    expire_time=self.settings.access_token_expire
                )
                await self.user_session_service.delete_for_device(
                    device_id=client_info.device_id,
                    user_id=user.id
```

```
                )
                user_session = await self.user_session_service.create(
                    user_id=user.id,
                    expire_time=self.settings.refresh_token_expire,
                    device_id=client_info.device_id
                )
            return SuccessLoginSchema(
                access_token=access_token,
                refresh_token=user_session.token,
                token_type="bearer",
                user=user,
            )

        except Exception as e:
            log.error("Error in login", email=email, exc_info=True)
            raise e


class RefreshTokensUseCase:
    def __init__(
        self,
        user_session_service: UserSessionServicePort,
        user_service: UserServicePort,
        jwt_processor: JWTProcessorPort,
        settings: Settings,
        session: AsyncSession
    ):
        self.user_session_service = user_session_service
        self.user_service = user_service
        self.jwt_processor = jwt_processor
        self.settings = settings

        self.__session = session

    async def __call__(
        self,
        refresh_token: str,
        client: RequestClient
    ) -> SuccessRefreshTokensSchema:
        async with self.__session.begin():
            user_session = await
self.user_session_service.get_by_rt(refresh_token)
            if not user_session:
                raise RefreshTokenNotExist
            now = datetime.now(timezone.utc)
            if user_session.expire_at < now:
                raise RefreshTokenExpired

            user = await
self.user_service.get_by_id(user_id=user_session.user_id)
            new_session = await
self.user_session_service.refresh_session(
                refresh_token=refresh_token,
```

```
                expire_time=self.settings.refresh_token_expire,
                device_id=client.device_id
            )
            access_token = self.jwt_processor.generate_access_token(
                user_data=user,
                expire_time=self.settings.access_token_expire
            )
            return SuccessRefreshTokensSchema(
                access_token=access_token,
                refresh_token=new_session.token,
            )


class SignUpUseCase:
    def __init__(
        self,
        user_service: UserServicePort,
        user_session_service: UserSessionServicePort,
        client_service: ClientServicePort,
        role_service: RoleServicePort,
        jwt_processor: JWTProcessorPort,
        settings: Settings,
        session: AsyncSession
    ):
        self.user_service = user_service
        self.user_session_service = user_session_service
        self.client_service = client_service
        self.role_service = role_service
        self.jwt_processor = jwt_processor
        self.settings = settings

        self.__session = session

    async def __call__(self, registration_data: SignUpRequest,
client_info: RequestClient) -> SuccessSignUpSchema:
        try:
            async with self.__session.begin():
                if await
self.user_service.exist_by_email(email=registration_data.email):
                    raise
UserAlreadyExist(email=registration_data.email)

                user = await self.user_service.create(
                    data=registration_data
                )
                client = await self.client_service.create(
                    user_id=user.id,
                )
                #await self.role_service.add_role(user_id=user.id,
role="user")
                await self.role_service.add_role(user_id=user.id,
role="client")
```

```
                user.roles += ["user", "client"]
                access_token =
self.jwt_processor.generate_access_token(
                    UserBaseSchema(
                        id=user.id,
                        roles=user.roles,
                    ),
                    expire_time=self.settings.access_token_expire
                )
                user_session = await self.user_session_service.create(
                    user_id=user.id,
                    expire_time=self.settings.refresh_token_expire,
                    device_id=client_info.device_id
                )
                return SuccessSignUpSchema(
                    access_token=access_token,
                    refresh_token=user_session.token,
                    token_type="bearer",
                    user=UserSchema(
                        id=user.id,
                        email=user.email,
                        roles=user.roles,
                        password_hash=user.password_hash
                    )
                )

        except Exception as e:
            log.error("Error on signup", data=registration_data,
exc_info=True)
            raise e

class LogoutUseCase:
    def __init__(
        self,
        user_session_service: UserSessionServicePort,
        session: AsyncSession
    ):
        self.user_session_service = user_session_service

        self.__session = session

    async def __call__(self, refresh_token: str) -> None:
        try:
            async with self.__session.begin():
                await
self.user_session_service.delete_one(refresh_token)

        except Exception as e:
            log.error("Error in LogoutUseCase", exc_info=True)



class LogoutAllUseCase:
```

```
    def __init__(
        self,
        user_session_service: UserSessionServicePort,
        jwt_processor: JWTProcessorPort,
        settings: Settings,
        session: AsyncSession
    ):
        self.user_session_service = user_session_service
        self.jwt_processor = jwt_processor
        self.settings = settings

        self.__session = session

    async def __call__(
        self,
        user: UserBaseSchema,
        client: RequestClient
    ) -> SuccessRefreshTokensSchema:
        # удалить все сеансы по user_id, кроме текущего
        try:
            async with self.__session.begin():
                await self.user_session_service.delete_all(user.id)
                user_session = await self.user_session_service.create(
                    user_id=user.id,
                    expire_time=self.settings.refresh_token_expire,
                    device_id=client.device_id
                )
                access_token =
self.jwt_processor.generate_access_token(
                    user_data=UserBaseSchema(
                        id=user.id,
                        roles=user.roles
                    ),
                    expire_time=self.settings.access_token_expire
                )
            return SuccessRefreshTokensSchema(
                refresh_token=user_session.token,
access_token=access_token
            )
        except Exception as e:
            log.error("Error on LogoutAllUseCase", exc_info=True)
            raise e
```

Файл /infrastructure/database/models.py

```
class TrainingActivity(Base):
    __tablename__ = "training_activity"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    name: Mapped[str] = mapped_column(String(128), nullable=False,
```

```
unique=True)
    description: Mapped[str | None] = mapped_column(Text,
nullable=True)
    duration_minutes: Mapped[int] = mapped_column(Integer,
nullable=False)
    max_group_size: Mapped[int] = mapped_column(Integer,
nullable=False)
    is_assessment: Mapped[bool] = mapped_column(Boolean,
nullable=False, default=False)
    is_active: Mapped[bool] = mapped_column(Boolean, nullable=False,
default=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    course_steps: Mapped[list["CourseStep"]] =
relationship("CourseStep", back_populates="activity", uselist=True)
    slot_templates: Mapped[list["SlotTemplate"]] =
relationship("SlotTemplate", back_populates="activity", uselist=True)
    slots: Mapped[list["TrainingSlot"]] = relationship("TrainingSlot",
back_populates="activity", uselist=True)


class NormativeDefinition(Base):
    __tablename__ = "normative_definition"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    name: Mapped[str] = mapped_column(String(128), nullable=False,
unique=True)
    unit: Mapped[str | None] = mapped_column(String(32),
nullable=True)
    value_type: Mapped[str] = mapped_column(String(32),
nullable=False, default="number")
    description: Mapped[str | None] = mapped_column(Text,
nullable=True)
    better_high: Mapped[bool] = mapped_column(Boolean, nullable=False,
default=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    results: Mapped[list["NormativeAttemptResult"]] = relationship(
        "NormativeAttemptResult",
        back_populates="normative",
        uselist=True,
    )


class Course(Base):
    __tablename__ = "course"
```

```
    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    name: Mapped[str] = mapped_column(String(128), nullable=False,
unique=True)
    description: Mapped[str | None] = mapped_column(Text,
nullable=True)
    duration_days: Mapped[int] = mapped_column(Integer,
nullable=False)
    min_rest_days: Mapped[int] = mapped_column(Integer,
nullable=False, default=1)
    max_gap_days: Mapped[int] = mapped_column(Integer, nullable=False,
default=5)
    is_active: Mapped[bool] = mapped_column(Boolean, nullable=False,
default=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    steps: Mapped[list["CourseStep"]] = relationship(
        "CourseStep",
        back_populates="course",
        uselist=True,
        cascade="all, delete-orphan",
        order_by="CourseStep.sequence_no",
    )
    enrollments: Mapped[list["CourseEnrollment"]] =
relationship("CourseEnrollment", back_populates="course",
uselist=True)


class CourseStep(Base):
    __tablename__ = "course_step"
    __table_args__ = (UniqueConstraint("course_id", "sequence_no",
name="uq_course_step_course_seq"),)

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
    course_id: Mapped[int] = mapped_column(Integer,
ForeignKey("course.id", ondelete="CASCADE"), nullable=False)
    activity_id: Mapped[int] = mapped_column(Integer,
ForeignKey("training_activity.id", ondelete="RESTRICT"),
nullable=False)
    sequence_no: Mapped[int] = mapped_column(Integer, nullable=False)
    title: Mapped[str | None] = mapped_column(String(128),
nullable=True)
    notes: Mapped[str | None] = mapped_column(Text, nullable=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())
```

```
    course: Mapped["Course"] = relationship("Course",
back_populates="steps", uselist=False)
    activity: Mapped["TrainingActivity"] =
relationship("TrainingActivity", back_populates="course_steps",
uselist=False)
    plan_items: Mapped[list["EnrollmentPlanItem"]] =
relationship("EnrollmentPlanItem", back_populates="course_step",
uselist=True)



class Client(Base):
    __tablename__ = "client"

    id: Mapped[int] = mapped_column(Integer, ForeignKey("user.id",
ondelete="CASCADE"), primary_key=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    user: Mapped["User"] = relationship("User",
back_populates="client", uselist=False)
    questionnaire: Mapped["ClientQuestionnaire"] = relationship(
        "ClientQuestionnaire",
        back_populates="client",
        uselist=False,
        cascade="all, delete-orphan",
    )
    enrollments: Mapped[list["CourseEnrollment"]] = relationship(
        "CourseEnrollment",
        back_populates="client",
        uselist=True,
        cascade="all, delete-orphan",
    )


class ClientQuestionnaire(Base):
    __tablename__ = "client_questionnaire"

    client_id: Mapped[int] = mapped_column(Integer,
ForeignKey("client.id", ondelete="CASCADE"), primary_key=True)
    birth_date: Mapped[date | None] = mapped_column(Date,
nullable=True)
    phone: Mapped[str | None] = mapped_column(String(32),
nullable=True)
    height_cm: Mapped[int | None] = mapped_column(Integer,
nullable=True)
    weight_kg: Mapped[float | None] = mapped_column(Numeric(5, 2),
nullable=True)
    emergency_contact_name: Mapped[str | None] =
mapped_column(String(128), nullable=True)
    emergency_contact_phone: Mapped[str | None] =
```

```
mapped_column(String(32), nullable=True)
    medical_notes: Mapped[str | None] = mapped_column(Text,
nullable=True)
    additional_data: Mapped[dict | None] = mapped_column(JSONB,
nullable=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    client: Mapped["Client"] = relationship("Client",
back_populates="questionnaire", uselist=False)


class Employee(Base):
    __tablename__ = "employee"

    id: Mapped[int] = mapped_column(Integer, ForeignKey("user.id",
ondelete="CASCADE"), primary_key=True)
    position_id: Mapped[int] = mapped_column(Integer,
ForeignKey("position.id", ondelete="RESTRICT"), nullable=False,
index=True)
    personnel_no: Mapped[str | None] = mapped_column(String(64),
nullable=True, unique=True)
    hired_at: Mapped[date | None] = mapped_column(Date, nullable=True)
    fired_at: Mapped[date | None] = mapped_column(Date, nullable=True)
    is_active: Mapped[bool] = mapped_column(Boolean, nullable=False,
default=True)
    created: Mapped[datetime] = mapped_column(DateTime(timezone=True),
default=func.now())
    updated: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    user: Mapped["User"] = relationship("User",
back_populates="employee", uselist=False)
    position: Mapped["Position"] = relationship("Position",
back_populates="employees", uselist=False)
    trainer: Mapped["Trainer"] = relationship(
        "Trainer",
        back_populates="employee",
        uselist=False,
        primaryjoin="Employee.id == foreign(Trainer.id)",
    )
```

Файл /infrastructure/di/providers.py

```
from src.infrastructure.core.settings import Settings, settings
from src.infrastructure.database.db import async_session_maker
from src.infrastructure.repositories.catalog_tc import
CatalogRepository
from src.infrastructure.repositories.client import ClientRepository
from src.infrastructure.repositories.content_tc import
ContentRepository
from src.infrastructure.repositories.enrollment_tc import
EnrollmentRepository
from src.infrastructure.repositories.personnel import
EmployeeRepository, PositionRepository
from src.infrastructure.repositories.role import RoleRepository
from src.infrastructure.repositories.schedule_tc import
ScheduleRepository
from src.infrastructure.repositories.trainer_tc import
TrainerRepository
from src.infrastructure.repositories.user import UserRepository
from src.infrastructure.repositories.user_session import
UserSessionRepository
from src.infrastructure.services.catalog_tc import CatalogService
from src.infrastructure.services.client import ClientService
from src.infrastructure.services.content_tc import ContentService
from src.infrastructure.services.enrollment_tc import
EnrollmentService
from src.infrastructure.services.mailing.mail import
MailingByMailService
from src.infrastructure.services.personnel import EmployeeService,
PositionService
from src.infrastructure.services.role import RoleService
from src.infrastructure.services.schedule_tc import ScheduleService
from src.infrastructure.services.security.jwt import JWTProcessor
from src.infrastructure.services.security.password import
PasswordProcessor
from src.infrastructure.services.trainer_tc import TrainerService
from src.infrastructure.services.user import UserService
from src.infrastructure.services.user_session import
UserSessionService
from src.infrastructure.taskiq.publisher import TaskiqPublisher


class SQLAlchemyProvider(Provider):
    @provide(scope=Scope.APP,
provides=async_sessionmaker[AsyncSession])
    def provide_session_maker(self) ->
async_sessionmaker[AsyncSession]:
        return async_session_maker

    @provide(scope=Scope.REQUEST, provides=AsyncSession)
    async def provide_session(self, session_maker:
```

```
async_sessionmaker[AsyncSession]) -> AsyncSession:
        async with session_maker() as session:
            yield session


class ServicesProvider(Provider):
    scope = Scope.REQUEST

    user_service = provide(UserService, provides=UserServicePort)
    user_session_service = provide(UserSessionService,
provides=UserSessionServicePort)
    client_service = provide(ClientService,
provides=ClientServicePort)
    role_service = provide(RoleService, provides=RoleServicePort)
    catalog_service = provide(CatalogService,
provides=CatalogServicePort)
    content_service = provide(ContentService,
provides=ContentServicePort)
    schedule_service = provide(ScheduleService,
provides=ScheduleServicePort)
    trainer_service = provide(TrainerService,
provides=TrainerServicePort)
    enrollment_service = provide(EnrollmentService,
provides=EnrollmentServicePort)
    position_service = provide(PositionService,
provides=PositionServicePort)
    employee_service = provide(EmployeeService,
provides=EmployeeServicePort)

    jwt_processor = provide(JWTProcessor, provides=JWTProcessorPort)
    password_processor = provide(PasswordProcessor,
provides=PasswordProcessorPort)

    mailing_by_mail_service = provide(MailingByMailService,
provides=MailingByMailServicePort)

    publisher = provide(TaskiqPublisher, provides=TaskPublisherPort)

class RepositoriesProvider(Provider):
    scope = Scope.REQUEST

    user_repo = provide(UserRepository, provides=UserRepositoryPort)
    user_session_repo = provide(UserSessionRepository,
provides=UserSessionRepositoryPort)
    client_repo = provide(ClientRepository,
provides=ClientRepositoryPort)
    role_repo = provide(RoleRepository, provides=RoleRepositoryPort)
    catalog_repo = provide(CatalogRepository,
provides=CatalogRepositoryPort)
    content_repo = provide(ContentRepository,
provides=ContentRepositoryPort)
    schedule_repo = provide(ScheduleRepository,
provides=ScheduleRepositoryPort)
```

```
    trainer_repo = provide(TrainerRepository,
provides=TrainerRepositoryPort)
    enrollment_repo = provide(EnrollmentRepository,
provides=EnrollmentRepositoryPort)
    position_repo = provide(PositionRepository,
provides=PositionRepositoryPort)
    employee_repo = provide(EmployeeRepository,
provides=EmployeeRepositoryPort)


class UseCasesProvider(Provider):
    scope = Scope.REQUEST

    login_uc = provide(LoginUseCase)
    logout_uc = provide(LogoutUseCase)
    logout_all_uc = provide(LogoutAllUseCase)
    refresh_tokens_uc = provide(RefreshTokensUseCase)
    signup_uc = provide(SignUpUseCase)

    create_position_uc = provide(CreatePositionUseCase)
    create_employee_uc = provide(CreateEmployeeUseCase)
    create_activity_uc = provide(CreateActivityUseCase)
    create_normative_uc = provide(CreateNormativeUseCase)
    create_course_uc = provide(CreateCourseUseCase)
    create_article_uc = provide(CreateArticleUseCase)
    update_article_uc = provide(UpdateArticleUseCase)
    create_weekly_slot_uc = provide(CreateWeeklySlotUseCase)
    duplicate_week_schedule_uc = provide(DuplicateWeekScheduleUseCase)
    delete_weekly_slot_uc = provide(DeleteWeeklySlotUseCase)
    create_trainer_uc = provide(CreateTrainerUseCase)
    upsert_questionnaire_uc = provide(UpsertQuestionnaireUseCase)
    enroll_on_course_uc = provide(EnrollOnCourseUseCase)
    update_trainer_profile_uc = provide(UpdateTrainerProfileUseCase)
    mark_attendance_uc = provide(MarkAttendanceUseCase)
    record_normative_results_uc =
provide(RecordNormativeResultsUseCase)
    mark_notification_read_uc = provide(MarkNotificationReadUseCase)
    mark_all_notifications_read_uc =
provide(MarkAllNotificationsReadUseCase)

    expire_session_uc = provide(ExpireSessionUseCase)

class SettingsProvider(Provider):
    scope = Scope.APP

    @provide
    def cfg(self) -> Settings:
        return settings
```

Файл /infrastructure/repositories/client.py

```
class ClientRepository(ClientRepositoryPort):
    def __init__(
        self,
        session: AsyncSession
    ):
        self.__session = session

    async def lock_client(self, client_id: int) -> None:
        stmt = select(Client.id).where(Client.id ==
client_id).with_for_update()
        await self.__session.execute(stmt)

    async def create(self, user_id: int) -> ClientSchema:
        client = Client(id=user_id)
        self.__session.add(client)
        await self.__session.flush()
        await self.__session.refresh(client)
        return ClientSchema(
            user_id=client.id
        )
```

Файл /infrastructure/repositories/trainer.py

```
class TrainerRepository(TrainerRepositoryPort):
    def __init__(self, session: AsyncSession):
        self._session = session

    async def create(self, data: TrainerCreate) -> TrainerRead:
        obj = Trainer(id=data.user_id, bio=data.bio,
experience_years=data.experience_years)
        self._session.add(obj)
        await self._session.flush()
        return await self.get(data.user_id)

    async def get(self, trainer_id: int) -> TrainerRead | None:
        stmt = (
            select(Trainer, User.first_name, User.last_name,
User.middle_name, Position.name)
            .join(User, Trainer.id == User.id)
            .outerjoin(Employee, Employee.id == Trainer.id)
            .outerjoin(Position, Position.id == Employee.position_id)
            .where(Trainer.id == trainer_id)
        )
        row = (await self._session.execute(stmt)).first()
        if row is None:
            return None
```

```
        trainer, first_name, last_name, middle_name, position_name =
row
        return TrainerRead(
            id=trainer.id,
            first_name=first_name,
            last_name=last_name,
            middle_name=middle_name,
            bio=trainer.bio,
            image=trainer.image,
            experience_years=trainer.experience_years,
            position_name=position_name,
        )

    async def list(self) -> list[TrainerRead]:
        stmt = (select(Trainer, User.first_name, User.last_name,
User.middle_name, Position.name)
            .join(User, Trainer.id == User.id)
            .outerjoin(Employee, Employee.id == Trainer.id)
            .outerjoin(Position, Position.id == Employee.position_id)
            .order_by(User.last_name, User.first_name))
        result = await self._session.execute(stmt)
        return [
            TrainerRead(
                id=trainer.id,
                first_name=first_name,
                last_name=last_name,
                middle_name=middle_name,
                bio=trainer.bio,
                image=trainer.image,
                experience_years=trainer.experience_years,
                position_name=position_name,
            )
            for trainer, first_name, last_name, middle_name,
position_name in result.all()
        ]

    async def update(self, trainer_id: int, data: TrainerUpdate) ->
None:
        payload = data.model_dump(exclude_unset=True)
        stmt = update(Trainer).where(Trainer.id ==
trainer_id).values(**payload)
        result = await self._session.execute(stmt)
        if result.rowcount == 0:
            raise TrainerNotFound
```

Файл /infrastructure/repositories/user_session.py

```
from datetime import datetime, timezone, timedelta

from sqlalchemy import select, delete
from sqlalchemy.ext.asyncio import AsyncSession

from src.application.core.exceptions.auth import RefreshTokenNotExist
from src.application.ports.repositories.user_session import
UserSessionRepositoryPort
from src.application.schemas.user_session import UserSessionSchema
from src.infrastructure.database.models import RefreshToken


class UserSessionRepository(UserSessionRepositoryPort):
    def __init__(
        self,
        session: AsyncSession,
    ):
        self.__session = session

    async def create(self, user_id: int, device_id: str, expire_time:
int) -> UserSessionSchema:
        expire_at = datetime.now(timezone.utc) +
timedelta(seconds=expire_time)
        user_session = RefreshToken(
            user_id=user_id,
            expire_at=expire_at,
            device_id=device_id
        )
        self.__session.add(user_session)
        await self.__session.flush()
        await self.__session.refresh(user_session)
        return UserSessionSchema(
            id=user_session.id,
            user_id=user_id,
            token=str(user_session.token),
            expire_at=user_session.expire_at,
            device_id=user_session.device_id
        )

    async def get_by_rt(self, refresh_token: str) -> UserSessionSchema
| None:
        query = select(
            RefreshToken.id,
            RefreshToken.user_id,
            RefreshToken.token,
            RefreshToken.device_id,
            RefreshToken.expire_at
```

```
        ).where(RefreshToken.token == refresh_token)

        result = await self.__session.execute(query)
        session_data = result.first()
        if session_data:
            return UserSessionSchema(
                id=session_data.id,
                user_id=session_data.user_id,
                token=str(session_data.token),
                expire_at=session_data.expire_at,
                device_id=session_data.device_id
            )

    async def refresh_session(self, refresh_token: str, expire_time:
int, device_id: str) -> UserSessionSchema:
        delete_query = (
            delete(RefreshToken)
            .where(RefreshToken.token == refresh_token)
            .returning(RefreshToken.user_id)
        )
        delete_result = await self.__session.execute(delete_query)
        returning_result = delete_result.first()
        if not returning_result:
            raise RefreshTokenNotExist
        new_session = await self.create(
            user_id=returning_result.user_id,
            expire_time=expire_time,
            device_id=device_id
        )
        return new_session

    async def delete_one(self, refresh_token: str) -> None:
        stmt = delete(RefreshToken).where(RefreshToken.token ==
refresh_token)
        await self.__session.execute(stmt)

    async def delete_all(self, user_id: int) -> None:
        stmt = delete(RefreshToken).where(RefreshToken.user_id ==
user_id)
        await self.__session.execute(stmt)

    async def delete_for_device(self, device_id: str, user_id: int):
        stmt = delete(RefreshToken).where(RefreshToken.user_id ==
user_id, RefreshToken.device_id == device_id)
        await self.__session.execute(stmt)

    async def delete_expired(self, quantity: int) -> None:
        stmt = delete(RefreshToken).where(
            RefreshToken.id.in_(
                select(RefreshToken.id)
                .where(RefreshToken.expire_at <
datetime.now(timezone.utc))
                .limit(quantity)
```

```
            )
        )
        await self.__session.execute(stmt)
```

Файл /infrastructure/repositories/content_tc.py

```
class ContentRepository(ContentRepositoryPort):
    def __init__(self, session: AsyncSession):
        self._session = session

    @staticmethod
    def _article_to_read(obj: Article) -> ArticleRead:
        return ArticleRead(
            id=obj.id,
            title=obj.title,
            summary=obj.summary,
            content=obj.content,
            is_published=obj.is_published,
            published_at=obj.published_at,
            created=obj.created,
            updated=obj.updated,
        )

    @staticmethod
    def _notification_to_read(obj: SiteNotification) ->
NotificationRead:
        return NotificationRead(
            id=obj.id,
            title=obj.title,
            message=obj.message,
            kind=obj.kind,
            payload=obj.payload,
            is_read=obj.is_read,
            read_at=obj.read_at,
            created=obj.created,
        )

    async def list_articles(self, published_only: bool = True) ->
list[ArticleRead]:
        stmt =
select(Article).order_by(Article.published_at.desc().nullslast(),
Article.created.desc())
        if published_only:
            stmt = stmt.where(Article.is_published.is_(True))
        result = await self._session.execute(stmt)
        return [self._article_to_read(item) for item in
result.scalars().all()]

    async def get_article(self, article_id: int, published_only: bool
= True) -> ArticleRead | None:
        stmt = select(Article).where(Article.id == article_id)
        if published_only:
```

```
            stmt = stmt.where(Article.is_published.is_(True))
        obj = (await self._session.execute(stmt)).scalar_one_or_none()
        return self._article_to_read(obj) if obj else None

    async def create_article(self, data: ArticleCreate) ->
ArticleRead:
        published_at = datetime.now(timezone.utc) if data.is_published
else None
        obj = Article(**data.model_dump(), published_at=published_at)
        self._session.add(obj)
        await self._session.flush()
        await self._session.refresh(obj)
        return self._article_to_read(obj)

    async def update_article(self, article_id: int, data:
ArticleUpdate) -> ArticleRead | None:
        obj = await self._session.get(Article, article_id)
        if obj is None:
            return None
        payload = data.model_dump(exclude_unset=True)
        if 'is_published' in payload and payload['is_published'] and
not obj.is_published:
            payload['published_at'] = datetime.now(timezone.utc)
        for key, value in payload.items():
            setattr(obj, key, value)
        await self._session.flush()
        await self._session.refresh(obj)
        return self._article_to_read(obj)

    async def list_notifications(self, user_id: int, unread_only: bool
= False) -> list[NotificationRead]:
        stmt = select(SiteNotification).where(SiteNotification.user_id
== user_id).order_by(SiteNotification.created.desc())
        if unread_only:
            stmt = stmt.where(SiteNotification.is_read.is_(False))
        result = await self._session.execute(stmt)
        return [self._notification_to_read(item) for item in
result.scalars().all()]

    async def get_notification_counters(self, user_id: int) ->
NotificationCounters:
        total_stmt =
select(func.count(SiteNotification.id)).where(SiteNotification.user_id
== user_id)
        unread_stmt =
select(func.count(SiteNotification.id)).where(SiteNotification.user_id
== user_id, SiteNotification.is_read.is_(False))
        total = int((await self._session.execute(total_stmt)).scalar()
or 0)
        unread = int((await
self._session.execute(unread_stmt)).scalar() or 0)
        return NotificationCounters(total=total, unread=unread)
```

```
    async def create_notification(self, user_id: int, title: str,
message: str, kind: str = "info", payload: dict | None = None) ->
NotificationRead:
        obj = SiteNotification(user_id=user_id, title=title,
message=message, kind=kind, payload=payload)
        self._session.add(obj)
        await self._session.flush()
        await self._session.refresh(obj)
        return self._notification_to_read(obj)

    async def mark_notification_read(self, user_id: int,
notification_id: int) -> bool:
        obj = await self._session.get(SiteNotification,
notification_id)
        if obj is None or obj.user_id != user_id:
            return False
        if not obj.is_read:
            obj.is_read = True
            obj.read_at = datetime.now(timezone.utc)
            await self._session.flush()
        return True

    async def mark_all_notifications_read(self, user_id: int) -> int:
        stmt = update(SiteNotification).where(SiteNotification.user_id
== user_id, SiteNotification.is_read.is_(False)).values(is_read=True,
read_at=datetime.now(timezone.utc))
        result = await self._session.execute(stmt)
        return result.rowcount or 0
```

Файл /infrastructure/taskiq/broker.py

```
class ExtendedNATSKeyValueScheduleSource(NATSKeyValueScheduleSource):
    def __init__(
        self,
        *args,
        kv_config: Optional[KeyValueConfig] = None,
        **kwargs,
    ) -> None:
        super().__init__(*args, **kwargs)
        self.kv_config = kv_config

    async def startup(self) -> None:
        self.nats_client = await nats.connect(
            servers=self.servers,
            **self.connect_options,
        )
        self.nats_jetstream = self.nats_client.jetstream()

        try:
            self.kv = await
self.nats_jetstream.key_value(self.bucket_name)
        except BucketNotFoundError:
```

```
            if self.kv_config is not None:
                self.kv = await self.nats_jetstream.create_key_value(
                    config=self.kv_config
                )
            else:
                self.kv = await self.nats_jetstream.create_key_value(
                    bucket=self.bucket_name,
                )

broker = PullBasedJetStreamBroker(
    servers=[settings.NATS_BROKER_URL],
    durable="main_worker",
    stream_name="background_tasks",
    subject="tasks",
    stream_config=StreamConfig(
        max_age=172800,
        retention=RetentionPolicy.WORK_QUEUE,
    )
)



source = ExtendedNATSKeyValueScheduleSource(
    servers=settings.NATS_BROKER_URL,
    kv_config=KeyValueConfig(
        bucket="taskiq_schedules",
        ttl=129600,
    ),
)

scheduler = TaskiqScheduler(
    broker=broker,
    sources=[source, LabelScheduleSource(broker)],
)

@broker.on_event(TaskiqEvents.WORKER_STARTUP)
async def startup(ctx) -> None:
    from src.infrastructure.di.setup import ioc_factory
    container = ioc_factory()
    setup_dishka(broker=broker, container=container)
```

Клиентская часть:

Файл /api/admin.ts

```
import { apiRequest } from './http';
import type {
  ActivityCreate,
  ActivityRead,
  CourseCreate,
  CourseRead,
  CourseShortRead,
  DuplicateWeekScheduleRequest,
  DuplicateWeekScheduleResult,
  NormativeCreate,
  NormativeRead,
  TrainerCreate,
  TrainerRead,
  TrainingSlotRead,
  WeeklyScheduleSlotCreate,
} from '../types/api';

export function listActivities() {
  return apiRequest<ActivityRead[]>('/admin/activities', { auth: true
});
}

export function createActivity(payload: ActivityCreate) {
  return apiRequest<ActivityRead>('/admin/activities', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}

export function listNormatives() {
  return apiRequest<NormativeRead[]>('/admin/normatives', { auth: true
});
}

export function createNormative(payload: NormativeCreate) {
  return apiRequest<NormativeRead>('/admin/normatives', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}

export function listAdminCourses() {
  return apiRequest<CourseShortRead[]>('/admin/courses', { auth: true
});
}

export function getAdminCourse(courseId: number) {
```

```
  return apiRequest<CourseRead>(`/admin/courses/${courseId}`, { auth:
true });
}

export function createCourse(payload: CourseCreate) {
  return apiRequest<CourseRead>('/admin/courses', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}

export function listTrainers() {
  return apiRequest<TrainerRead[]>('/admin/trainers', { auth: true });
}

export function createTrainer(payload: TrainerCreate) {
  return apiRequest<TrainerRead>('/admin/trainers', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}

export function listWeeklySchedule(weekStart: string) {
  return apiRequest<TrainingSlotRead[]>(`/admin/weekly-
schedule?week_start=${weekStart}`, { auth: true });
}

export function createWeeklySlot(payload: WeeklyScheduleSlotCreate) {
  return apiRequest<TrainingSlotRead>('/admin/weekly-schedule/slots',
{
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
  });
}

export function deleteWeeklySlot(slotId: number) {
  return apiRequest<void>(`/admin/weekly-schedule/slots/${slotId}`, {
    method: 'DELETE',
    auth: true,
  });
}

export function duplicateWeekSchedule(payload:
DuplicateWeekScheduleRequest) {
  return apiRequest<DuplicateWeekScheduleResult>('/admin/weekly-
schedule/duplicate', {
    method: 'POST',
    auth: true,
    body: JSON.stringify(payload),
```

```
  });
}
```

Файл /api/auth.ts

```
import { API_BASE_URL } from '../config';
import { getDeviceId } from '../lib/device';
import { apiRequest } from './http';
import type { AuthResponse, LoginRequest, SignUpRequest } from
'../types/api';

let refreshPromise: Promise<AuthResponse | null> | null = null;

function parseRefreshResponse(response: Response):
Promise<AuthResponse> {
  return response.json() as Promise<AuthResponse>;
}

export function signIn(payload: LoginRequest) {
  return apiRequest<AuthResponse>('/auth/signin', {
    method: 'POST',
    body: JSON.stringify(payload),
  });
}

export function signUp(payload: SignUpRequest) {
  return apiRequest<AuthResponse>('/auth/signup', {
    method: 'POST',
    body: JSON.stringify(payload),
  });
}

export async function refreshTokens(): Promise<AuthResponse | null> {
  if (refreshPromise) {
    return refreshPromise;
  }

  refreshPromise = fetch(`${API_BASE_URL}/auth/refresh`, {
    method: 'POST',
    credentials: 'include',
    headers: {
      'X-Device-ID': getDeviceId(),
    },
  })
    .then(async (response) => {
      if (response.status === 401) {
        return null;
      }

      if (!response.ok) {
```

```
        let detail = 'Не удалось обновить сессию';
        try {
          const payload = await response.json();
          if (typeof payload?.detail === 'string') {
            detail = payload.detail;
          }
        } catch {
          // ignore parse error
        }
        throw new Error(detail);
      }

      return parseRefreshResponse(response);
    })
    .finally(() => {
      refreshPromise = null;
    });

  return refreshPromise;
}

export function logout() {
  return apiRequest<{ result: string }>('/auth/logout', {
    method: 'POST',
    auth: true,
  });
}

export function logoutAll() {
  return apiRequest<AuthResponse>('/auth/logout-all', {
    method: 'POST',
    auth: true,
  });
}
```

Файл /components/AppShell.tsx

```
import { NavLink, Outlet, useLocation } from 'react-router-dom';
import { GlobalHeader } from './GlobalHeader';
import { useAuth } from '../hooks/useAuth';

function getCabinetLinks(pathname: string, roles: string[]) {
  if (pathname.startsWith('/app/admin')) {
    return [
      { to: '/app/admin', label: 'Обзор' },
      { to: '/app/admin/schedule', label: 'Расписание недели' },
      { to: '/app/admin/courses', label: 'Курсы' },
      { to: '/app/admin/activities', label: 'Активности' },
      { to: '/app/admin/normatives', label: 'Нормативы' },
      { to: '/app/admin/trainers', label: 'Тренеры' },
      { to: '/app/admin/articles', label: 'Статьи' },
```

```
    ];
  }
  if (pathname.startsWith('/app/trainer')) {
    return [
      { to: '/app/trainer', label: 'Обзор' },
      { to: '/app/trainer/schedule', label: 'Расписание по дням' },
      { to: '/app/trainer/profile', label: 'Мои данные' },
    ];
  }
  if (pathname.startsWith('/app/client')) {
    return [
      { to: '/app/client', label: 'Обзор' },
      { to: '/app/client/profile', label: 'Моя анкета' },
      { to: '/app/client/enrollments', label: 'Мои курсы' },
      { to: '/app/client/plan', label: 'Расписание недели' },
      { to: '/app/client/normatives', label: 'Нормативы' },
    ];
  }

  if (roles.includes('admin')) return [{ to: '/app/admin', label:
'Кабинет администратора' }];
  if (roles.includes('trainer')) return [{ to: '/app/trainer', label:
'Кабинет тренера' }];
  return [{ to: '/app/client', label: 'Кабинет клиента' }];
}

export function AppShell() {
  const location = useLocation();
  const { user } = useAuth();
  const links = getCabinetLinks(location.pathname, user?.roles ?? []);

  return (
    <div className="app-layout">
      <GlobalHeader />
      <main className="app-layout__main">
        <div className="cabinet-tabs">
          {links.map((link) => (
            <NavLink key={link.to} to={link.to} end={link.to ===
'/app/client' || link.to === '/app/trainer' || link.to ===
'/app/admin'} className={({ isActive }) => isActive ? 'cabinet-tab
cabinet-tab--active' : 'cabinet-tab'}>
              {link.label}
            </NavLink>
          ))}
        </div>
        <div className="page-content">
          <Outlet />
        </div>
      </main>
    </div>
  );
}
```

Файл /components/StatCard.tsx

```
export function StatCard({ label, value, hint }: { label: string;
value: string | number; hint?: string }) {
  return (
    <div className="stat-card">
      <div className="stat-card__label">{label}</div>
      <div className="stat-card__value">{value}</div>
      {hint ? <div className="stat-card__hint">{hint}</div> : null}
    </div>
  );
}
```

Файл /context/AuthContext.tsx

```
import { createContext, useCallback, useEffect, useMemo, useState,
type ReactNode } from 'react';
import { logout as logoutRequest, refreshTokens, signIn as
signInRequest, signUp as signUpRequest } from '../api/auth';
import { configureApiAuth, setAccessToken } from '../api/http';
import { getUserFromToken, hasAnyRole, type SessionUser } from
'../lib/jwt';
import type { LoginRequest, Role, SignUpRequest } from '../types/api';

interface AuthContextValue {
  user: SessionUser | null;
  accessToken: string | null;
  isAuthenticated: boolean;
  isBootstrapping: boolean;
  signIn: (payload: LoginRequest) => Promise<void>;
  signUp: (payload: SignUpRequest) => Promise<void>;
  logout: () => Promise<void>;
  hasRole: (...roles: Role[]) => boolean;
  refreshSession: () => Promise<string | null>;
}

export const AuthContext = createContext<AuthContextValue |
null>(null);

function applyToken(token: string | null, setToken: (value: string |
null) => void, setUser: (value: SessionUser | null) => void) {
  setAccessToken(token);
  setToken(token);
  setUser(getUserFromToken(token));
}

export function AuthProvider({ children }: { children: ReactNode }) {
  const [accessTokenState, setAccessTokenState] = useState<string |
null>(null);
  const [user, setUser] = useState<SessionUser | null>(null);
  const [isBootstrapping, setIsBootstrapping] = useState(true);
```

```
  const clearSession = useCallback(() => {
    applyToken(null, setAccessTokenState, setUser);
  }, []);

  const refreshSession = useCallback(async () => {
    try {
      const response = await refreshTokens();
      const token = response?.access_token ?? null;
      applyToken(token, setAccessTokenState, setUser);
      return token;
    } catch {
      clearSession();
      return null;
    }
  }, [clearSession]);

  useEffect(() => {
    configureApiAuth({
      refresh: refreshSession,
      onUnauthorized: clearSession,
    });
  }, [refreshSession, clearSession]);

  useEffect(() => {
    let isMounted = true;

    void (async () => {
      try {
        await refreshSession();
      } finally {
        if (isMounted) {
          setIsBootstrapping(false);
        }
      }
    })();

    return () => {
      isMounted = false;
    };
  }, [refreshSession]);

  const signIn = useCallback(async (payload: LoginRequest) => {
    const response = await signInRequest(payload);
    applyToken(response.access_token, setAccessTokenState, setUser);
  }, []);

  const signUp = useCallback(async (payload: SignUpRequest) => {
    const response = await signUpRequest(payload);
    applyToken(response.access_token, setAccessTokenState, setUser);
  }, []);

  const logout = useCallback(async () => {
    try {
```

```
      if (accessTokenState) {
        await logoutRequest();
      }
    } finally {
      clearSession();
    }
  }, [accessTokenState, clearSession]);

  const hasRoleFn = useCallback(
    (...roles: Role[]) => {
      if (!user) return false;
      return hasAnyRole(user.roles, roles);
    },
    [user],
  );

  const value = useMemo<AuthContextValue>(
    () => ({
      user,
      accessToken: accessTokenState,
      isAuthenticated: Boolean(user),
      isBootstrapping,
      signIn,
      signUp,
      logout,
      hasRole: hasRoleFn,
      refreshSession,
    }),
    [user, accessTokenState, isBootstrapping, signIn, signUp, logout,
hasRoleFn, refreshSession],
  );

  return <AuthContext.Provider
value={value}>{children}</AuthContext.Provider>;
}



import { useContext } from 'react';
import { AuthContext } from '../context/AuthContext';

export function useAuth() {
  const value = useContext(AuthContext);
  if (!value) {
    throw new Error('useAuth must be used within AuthProvider');
  }
  return value;
}
```

Файл /pages/admin/ActivitiesPage.tsx

```
import { useEffect, useState } from 'react';
import { createActivity, listActivities } from '../../api/admin';
import { ApiError } from '../../api/http';
import { SectionCard } from '../../components/SectionCard';
import type { ActivityCreate, ActivityRead } from '../../types/api';

const initialForm: ActivityCreate = {
  name: '',
  description: '',
  duration_minutes: 60,
  max_group_size: 10,
  is_assessment: false,
  is_active: true,
};

export function AdminActivitiesPage() {
  const [items, setItems] = useState<ActivityRead[]>([]);
  const [form, setForm] = useState<ActivityCreate>(initialForm);
  const [error, setError] = useState<string | null>(null);
  const [message, setMessage] = useState<string | null>(null);

  async function load() {
    try {
      setItems(await listActivities());
    } catch (error) {
      setError(error instanceof ApiError ? error.detail : 'Не удалось
загрузить активности');
    }
  }

  useEffect(() => {
    void load();
  }, []);

  async function handleSubmit(event: React.FormEvent<HTMLFormElement>)
{
    event.preventDefault();
    setError(null);
    setMessage(null);
    try {
      await createActivity(form);
      setMessage('Активность создана.');
      setForm(initialForm);
      await load();
    } catch (error) {
      setError(error instanceof ApiError ? error.detail : 'Не удалось
создать активность');
    }
  }

  return (
```

```
    <div className="page-stack">
      <div className="page-
header"><div><h1>Активности</h1><p>Справочник типов занятий,
используемых при составлении курсов.</p></div></div>
      {error ? <div className="alert alert--error">{ }</div> : null}
      {message ? <div className="alert alert--success">{ }</div> :
null}

      <div className="grid grid--two-admin">
        <SectionCard title="Создать активность">
          <form className="form-grid" onSubmit={handleSubmit}>
            <label className="field field--
full"><span>Название</span><input value={form.name} onChange={(event)
=> setForm((prev) => ({ ...prev, name: event.target.value }))}
required /></label>
            <label className="field field--
full"><span>Описание</span><textarea rows={4} value={form.description
?? ''} onChange={(event) => setForm((prev) => ({ ...prev, description:
event.target.value }))} /></label>
            <label className="field"><span>Длительность,
мин</span><input type="number" min="1" value={form.duration_minutes}
onChange={(event) => setForm((prev) => ({ ...prev, duration_minutes:
Number(event.target.value) }))} required /></label>
            <label className="field"><span>Лимит группы</span><input
type="number" min="1" value={form.max_group_size} onChange={(event) =>
setForm((prev) => ({ ...prev, max_group_size:
Number(event.target.value) }))} required /></label>
            <label className="checkbox-field"><input type="checkbox"
checked={Boolean(form.is_assessment)} onChange={(event) =>
setForm((prev) => ({ ...prev, is_assessment: event.target.checked }))}
/> Это норматив / оценочное занятие</label>
            <label className="checkbox-field"><input type="checkbox"
checked={Boolean(form.is_active)} onChange={(event) => setForm((prev)
=> ({ ...prev, is_active: event.target.checked }))} /> Активно</label>
            <div className="field field--full"><button
className="button" type="submit">Создать</button></div>
          </form>
        </SectionCard>

        <SectionCard title="Список активностей">
          <div className="table-wrap">
            <table className="table">

<thead><tr><th>ID</th><th>Название</th><th>Минут</th><th>Группа</th><t
h>Флаги</th></tr></thead>
              <tbody>
                {items.map((item) => (
                  <tr key={item.id}>
                    <td>{item.id}</td>
                    <td><div>{item.name}</div><div
className="muted">{item.description || '—'}</div></td>
                    <td>{item.duration_minutes}</td>
                    <td>{item.max_group_size}</td>
```

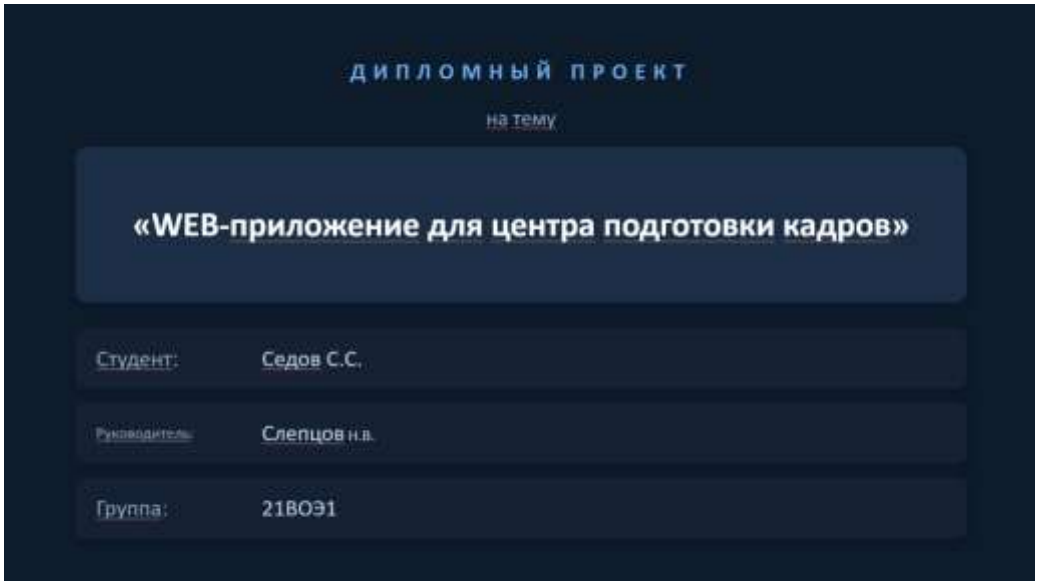
```
                    <td>{item.is_assessment ? 'assessment' :
'regular'} / {item.is_active ? 'active' : 'disabled'}</td>
                  </tr>
                ))}
              </tbody>
            </table>
          </div>
        </SectionCard>
      </div>
    </div>
  );
}
```

Приложение В

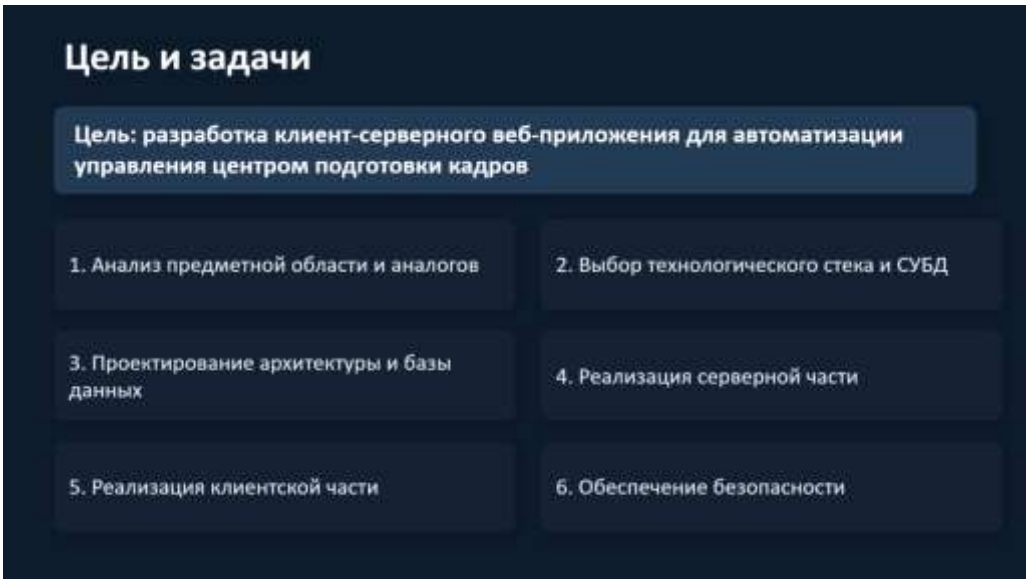
(обязательное)

WEB-приложение для центра подготовки кадров

Презентация


ДИПЛОМНЫЙ ПРОЕКТ
на тему
«WEB-приложение для центра подготовки кадров»
Студент: Седов С.С.
Руководитель: Слепцов н.в.
Группа: 21ВОЭ1

Актуальность
Проблема
Учёт клиентов, расписание и нормативы ведутся вручную — в таблицах и бумажных журналах
Нет единой системы для клиента, тренера и администратора
Высокий риск ошибок и потери данных
Отсутствие сервиса, предоставляющего курсы по разным направлениям физ. подготовки
Решение
Единая веб-платформа для клиента, тренера и администратора
Подготовительные курсы различных направлений
Ведение нормативов и посещаемости в цифровом виде


Цель и задачи
Цель: разработка клиент-серверного веб-приложения для автоматизации управления центром подготовки кадров
1. Анализ предметной области и аналогов
2. Выбор технологического стека и СУБД
3. Проектирование архитектуры и базы данных
4. Реализация серверной части
5. Реализация клиентской части
6. Обеспечение безопасности

Анализ аналогов

Прямых аналогов разрабатываемой системы не существует. Рассматриваемые решения относятся к системам управления фитнес-центрами или военной подготовкой.

uni-gym.ru

Фитнес-центр

✓ Расписание, бесплатное пробное занятие, информация о тренерах

✗ Отсутствие личного кабинета, нет ведения показателей клиентов

alexfitness.ru

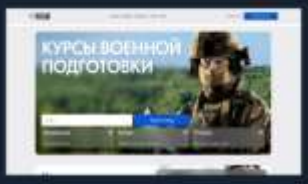
Фитнес-центр

✓ Просмотр абонементов, онлайн-запись, расписание, наличие приложения

✗ Отсутствие контроля показателей клиентов

утцзащитник.рф

Военная подготовка



✓ Различные курсы, опытные преподаватели, тренировки на тактических площадках

✗ Отсутствие личного кабинета, нет онлайн-записи

Технологический стек

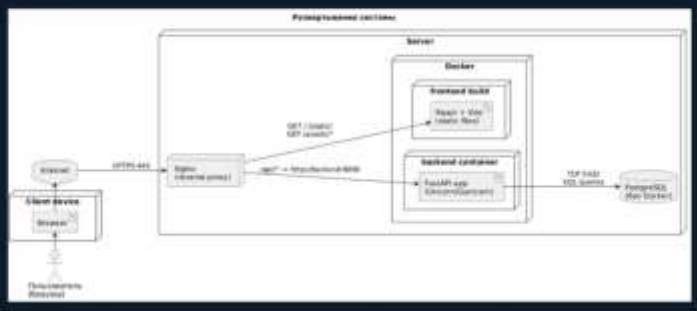
Серверная часть

- Python 3.12
- FastAPI
- SQLAlchemy + Alembic
- PostgreSQL
- NATS + TaskIQ
- Dishka (DI)
- PyJWT
- Docker
- Nginx

Клиентская часть

- TypeScript
- React 18
- Vite
- React Router v6
- CSS

Архитектура системы



Проектирование базы данных

Архитектура серверной части

/api Слой представления	Роутеры FastAPI, обработчики HTTP-запросов, валидация через Pydantic, проверка ролей
/application Прикладной слой	UseCase — сценарии бизнес-логики с атомарными транзакциями, схемы Pydantic, Порты (Protocol)
/infrastructure Инфраструктурный слой	Реализации сервисов и репозиториев. ORM-модели SQLAlchemy, Провайдеры Dishka, настройки, логгирование, фоновые задачи

Архитектура клиентской части

/types
TypeScript-интерфейсы, соответствующие схемам бэкенда

/api
HTTP-клиент с авто-обновлением токена. Модули по группам эндпоинтов

/context
AuthContext — хранение токена в памяти, управление сессией

/components
ProtectedRoute, AppShell, GlobalHeader — переиспользуемые компоненты

/pages
Страницы сгруппированы по ролям: auth, public, client, trainer, admin

App.tsx
React Router — маршрутизация с защитой по ролям

Безопасность

Хеширование паролей

bcrypt через passlib.

JWT-аутентификация

Access-токен в памяти (защита от XSS). Refresh-токен в httpOnly cookie (защита от кражи). Алгоритм RS256 (асимметричные ключи)

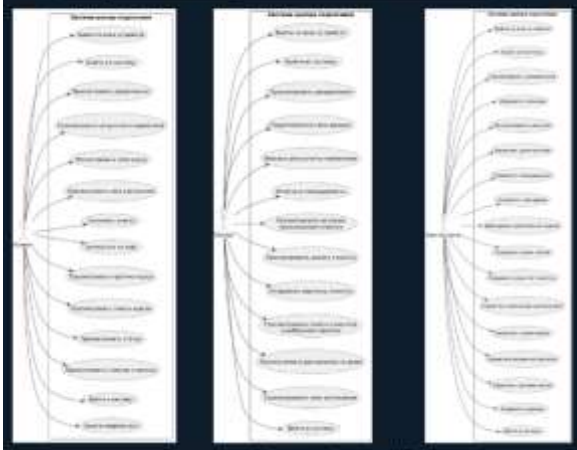
Разграничение ролей

Роли: client, trainer, admin. Проверка на уровне роутера (RoleRequired) и на уровне клиента (ProtectedRoute)

HTTPS

SSL-сертификат Let's Encrypt. Nginx терминирует TLS.

Варианты использования



Клиент

Регистрация, запись на курс, просмотр расписания, анкета, нормативы

Тренер

Расписание по дням, отметка посещаемости, внесение нормативов, карточки клиентов

Администратор

Управление курсами, расписанием, тренерами, активностями, статьями

Роль определяется JWT-токеном и проверяется на уровне API и клиента

Заключение

Результаты работы

- Разработано клиент-серверное веб-приложение с тремя ролями пользователей
- Реализовано автоматическое формирование расписания при записи на курс
- Настроены фоновые задачи через NATS + TaskIQ
- Обеспечена безопасность: JWT (RS256), bcrypt, HTTPS, разграничение ролей
- Система развёрнута в Docker с Nginx и SSL-сертификатом Let's Encrypt

Перспективы развития

- Мобильное приложение для iOS и Android
- Интеграция с платёжными системами
- Рассылка уведомлений в мессенджеры
- Аналитика и отчётность для администратора