|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ \_\_\_\_\_\_ «Информатика и системы управления» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_\_ «Теоретическая информатика и компьютерные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ***

***НА ТЕМУ:***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Студент \_\_ИУ9-82Б\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** В.В. Потребина

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель ВКР **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Нормоконтролер **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2025 г.*

АННОТАЦИЯ

Расчетно-пояснительная записка к выпускной квалификационной работе на тему «Разработка веб-приложения системы управления базой знаний для студентов с применением технологий искусственного интеллекта» состоит из 61 страниц. Работа включает в себя 21 листинг, 13 таблиц и 16 рисунков. В процессе выполнения *было* использовано 12 источников.

Цель работы заключается в разработке веб-приложения для студентов, предоставляющее удобный доступ к важной информации и ресурсам, а также включающее в себя ассистента на основе технологий искусственного интеллекта.

Структура работы включает в себя 4 главы: постановка задачи и анализ предметной области (глава 1), проектирование системы (глава 2), реализация системы (глава 3) и тестирование (глава 4).

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc200671542)

[1. Постановка задачи и анализ предметной области 6](#_Toc200671543)

[2. Проектирование системы 7](#_Toc200671544)

[2.1. Архитектура и интерфейс веб-приложения 7](#_Toc200671545)

[2.1.1. Предметно-ориентированное проектирование 8](#_Toc200671546)

[2.1.2. Архитектура клиентской части 13](#_Toc200671547)

[2.1.3. Клиент-серверное взаимодействие 15](#_Toc200671548)

[2.2. Проектирование базы данных 16](#_Toc200671549)

[2.2.1. ER-модель базы данных 16](#_Toc200671550)

[2.2.2. Преобразование ER-модели в реляционную модель 18](#_Toc200671551)

[2.3. Взаимодействие с большими языковыми моделями 23](#_Toc200671552)

[3. Реализация системы 27](#_Toc200671553)

[3.1. Выбор стека технологий 27](#_Toc200671554)

[3.2. Схема программного обеспечения 28](#_Toc200671555)

[3.3. Серверная часть 29](#_Toc200671556)

[3.3.1. Инфраструктурный уровень 29](#_Toc200671557)

[3.3.2. Доменный уровень 36](#_Toc200671558)

[3.3.3. Прикладной уровень 37](#_Toc200671559)

[3.3.4. Уровень представления 38](#_Toc200671560)

[3.4. Клиентская часть 40](#_Toc200671561)

[3.4.1. Атомная архитектура 40](#_Toc200671562)

[3.4.2. Обработка API 43](#_Toc200671563)

[3.4.3. Обработка состояния приложения 44](#_Toc200671564)

[3.4.4. Провайдеры 46](#_Toc200671565)

[3.5. Хранение файлов 46](#_Toc200671566)

[3.6. Искусственный интеллект 49](#_Toc200671567)

[4. Тестирование и анализ результатов 51](#_Toc200671568)

[ЗАКЛЮЧЕНИЕ 59](#_Toc200671569)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 60](#_Toc200671570)

# ВВЕДЕНИЕ

В сфере высшего образования уже давно применяются информационные системы, обеспечивая хранение учебных материалов, удобный доступ к ним студентов, мониторинг успеваемости студентов и поддержку в планировании учебного процесса. Особенно актуально их использование систем в вузах, где объем учебной информации (лекций, семинаров, лабораторных работ) достигает значительных размеров.

Наряду с централизацией всех накопленных знаний, довольно сложной задачей является обеспечение эффективного доступа к этим материалам. Традиционные подходы включают индексацию содержимого, многоуровневый поиск, каталогизацию по различным критериям, присвоение тегов и облако тегов. Однако с конца 2022 года благодаря развитию нейросетевых технологий, особенно моделей типа трансформер, появился качественно новый инструмент для работы с большими объёмами данных [1].

Сегодня уже очевидно, что студенты активно используют нейросетевые модели для самообучения и понимания сложных тем. Проблема состоит в том, что большинство таких моделей обучаются на открытых источниках, нередко содержащих нерелевантную и даже непроверенную информацию, что делает их применение в образовательном процессе без дополнительной адаптации небезопасным.

Решением этой проблемы может стать специальное приложения, интегрирующее нейросетевую модель с алгоритмом формирования контекста на основе материалов конкретного университета. [2] Такой подход позволяет создать собственный датасет, основанный на официально утверждённых материалах кафедры. В результате модель сможет формировать ответы на запросы студентов, опираясь не на случайные интернет-источники, а на официально утверждённые учебные материалы кафедры, что необходимо для успешного освоения учебной программы.

# 1. Постановка задачи и анализ предметной области

Цель дипломной работы – спроектировать и разработать веб-приложение для студентов кафедры ИУ9 МГТУ имени Н.Э. Баумана, предоставляющее удобный доступ к важной учебной информации и ресурсам, которые способствуют повышению эффективности образовательного процесса.

Для достижения цели дипломной работы были поставлены следующие задачи:

1. Изучить и проанализировать предметную область, последние достижения в области применения ИИ.
2. Разработать архитектуру веб-приложения, спроектировать инфологическую модель базы данных, разработать алгоритмы аутентификации и авторизации пользователей, проанализировать существующие возможности по дообучению больших языковых моделей.
3. Реализовать веб-приложение согласно выбранной архитектуре, создать интуитивно понятный интерфейс, разработать серверную части, включая аутентификацию пользователей и обеспечивать безопасность данных, а также реализовать хранилище данных на основе разработанной ER-модели. Реализовать алгоритмы интеграции с сервисом OpenAI для предоставления расширенных возможностей приложения.
4. Наполнить систему реальными данными и провести тестирование для подтверждения работоспособности системы и выполнения заявленных функций.

# 2. Проектирование системы

## 2.1. Архитектура и интерфейс веб-приложения

Одной из ключевых задач при создании программного обеспечения, особенно в области веб-разработки, является обеспечение его надёжности, расширяемости и удобства сопровождения. Эти качества напрямую зависят от того, насколько грамотно спроектирована архитектура системы. На практике часто можно столкнуться с ситуацией, когда изначально слабо продуманная архитектура приводит к трудностям при внедрении новых функций, усложнению отладки и снижению производительности. Кроме того, ошибки архитектурного уровня гораздо сложнее исправлять на более поздних этапах, особенно когда проект уже перешёл в стадию реального использования

Типичными проблемами, которые возникают при отсутствии чётко определённой архитектурной модели, является недостаточная модульность системы. Это существенно затрудняет повторное использование отдельных компонентов и усложняет дальнейшее развитие проекта. Кроме того, системы, разработанные без продуманной архитектуры, часто характеризуются низкой масштабируемостью, особенно критичной при увеличении количества пользователей, когда требования к производительности и ресурсам существенно возрастают.

Ещё одной распространённой проблемой является сложность проведения тестирования, поскольку логика приложения оказывается тесно связанной с представлением данных или их хранением, что делает затруднительным выявление и исправление ошибок. Помимо этого, отсутствие чётких границ между компонентами приводит к их сильной взаимозависимости, а значит, и к высокой связности, усложняя процессы рефакторинга и поддержки.

Также недостаточно структурированная архитектура негативно влияет на безопасность системы, поскольку затрудняет централизованное управление доступом и проведение необходимых проверок, увеличивая риски утечки данных и других уязвимостей.

Кроме технических сложностей, страдает и пользовательский опыт: медленная работа приложения, нестабильность при нагрузках, и непродуманная структура интерфейса становятся серьёзными барьерами для использования системы в образовательном процессе.

Поэтому одним из самых важных этапов разработки веб-приложения является проектирование архитектуры, так как именно она закладывает фундамент для всей последующей реализации, определяет распределение ответственности между компонентами и обеспечивает долгосрочную надёжность и удобство сопровождения системы [3].

### 2.1.1. Предметно-ориентированное проектирование

Одним из самых популярных подходов, применяемых при проектировании серверной части приложения, является предметно-ориентированное проектирование (Domain Driven Design - DDD) [4].

DDD представляет собой совокупность принципов, паттернов и методик, ориентированных на глубокое осмысление и моделирование предметной области. Ключевая идея заключается в том, что концептуальная модель и программная реализация должны непосредственно отражать реальные бизнес-процессы и сущности, а не быть механической проекцией технических или инфраструктурных ограничений.

В объектно-ориентированной программе код для реализации интерфейса пользователя, обращений к базе данных и других технических задач нередко вписывают напрямую в объекты прикладной модели. Кроме того, часть прикладной логики, т.е. алгоритмической части программы, часто реализуется прямо в элементах интерфейса пользователя и сценариях баз данных. Это происходит потому, что так легче всего работать на ближнюю перспективу. Когда код, относящийся к операциям предметной области, распределен по огромным объемам другого кода, его становится очень трудно разыскивать и анализировать. Поверхностные изменения в интерфейсе пользователя могут случайно затронуть и операции алгоритмической части. А чтобы изменить какие-либо правила делового регламента, потребуется тщательная трассировка кода интерфейса, обращений к базе данных и других элементов программы. Реализация логически последовательных, основанных на модели объектов, становится нецелесообразной. Автоматизированное тестирование оказывается малоэффективным. Учитывая объем технологий и операций в каждом из этих видов работ, структуру программы следует поддерживать в максимально упрощенном виде, иначе ее понимание станет невозможным.

Разработка программ, которые могут выполнять сложные задачи, требует разделения обязанностей, которое бы позволило сосредоточиться на принципиальных частях архитектуры в отдельности, в изоляции от других. В то же время, сложные взаимосвязи в пределах системы необходимо поддерживать, несмотря на разделение. Существует множество способов разделения программной системы, но преимущество отдается многоуровневой архитектуре, состоящей из нескольких достаточно стандартных уровней в соответствии с рисунком 1 [5].

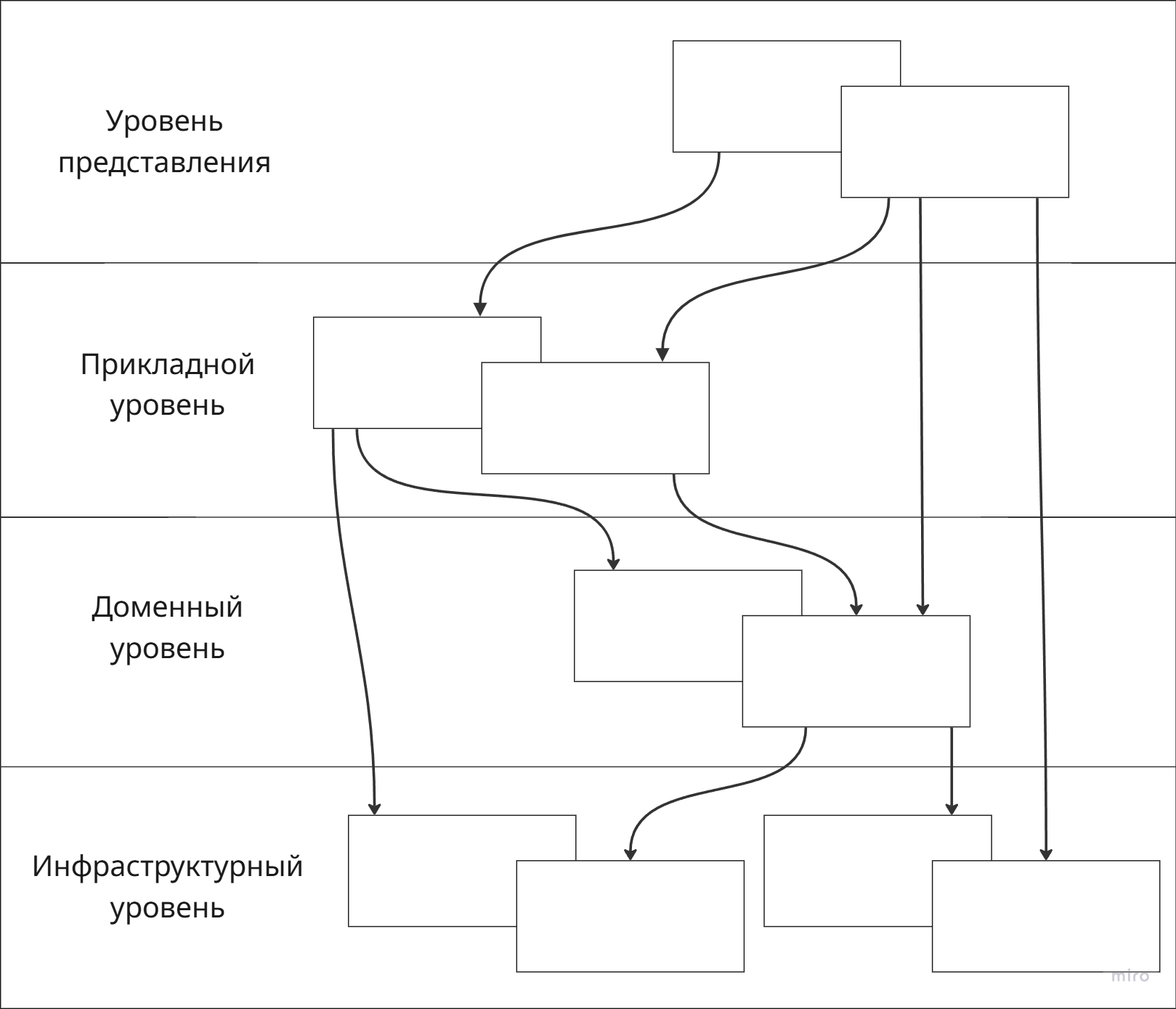


Рисунок 1 – Уровни архитектуры

Ценность многоуровневости состоит в том, что каждый уровень специализируется на конкретном аспекте программы. Такая специализация позволяет выполнять более связное проектирование каждого аспекта, и получающуюся архитектуру намного легче интерпретировать. Рассмотрим более подробно каждый из уровней.

Уровень представления отвечает за взаимодействие внешних факторов с системой, будь то человек или другая компьютерная программа. Основная задача – отображение информации и обработка пользовательских команд, включая действия, инициируемые через пользовательский интерфейс, API или сообщения из внешней системы.

Важной особенностью является то, что данный уровень не должен содержать бизнес-логики. Он лишь интерпретирует команды пользователя и передаёт их в прикладной слой для дальнейшей обработки. Это способствует чистоте архитектуры и упрощает сопровождение UI-компонентов, позволяя менять внешний вид или протоколы взаимодействия без затрагивания внутренних слоёв. Например, REST-контроллер принимает HTTP-запрос, извлекает параметры и инициирует соответствующий сценарий на уровне прикладных операций.

Показательной иллюстрацией может служить пользователь, заполняющий форму на веб-сайте для создания заказа. Система отображает соответствующий интерфейс, принимает данные формы и передаёт их на следующий уровень. Важно подчеркнуть, что бизнес-логика, например, валидация допустимости заказа, расчёт итоговой стоимости на этом этапе не реализуется, что способствует соблюдению принципа разделения ответственности.

Прикладной уровень (также его называют сервисный уровень) реализует сценарии использования системы: он определяет, какие действия должны быть выполнены при тех или иных запросах, и координирует работу объектов, представляющих бизнес-сущности. Его задача заключается не столько в реализации бизнес-правил, сколько в организации и координации бизнес- процессов. Этот уровень определяет, какие действия должны быть выполнены в ответ на команду пользователя, и каким образом они должны быть делегированы объектам доменной модели.

Благодаря такому разграничению прикладной слой становится независимым от специфики UI и инфраструктуры, что облегчает тестирование и повторное использование сценариев.

Таким образом, уровень прикладных операций формирует «каркас» выполнения задачи, чётко координируя взаимодействие между слоями, но при этом оставаясь независимым от конкретной реализации бизнес-логики или инфраструктурных деталей.

Доменный уровень представляет собой ядро архитектуры. Здесь сосредоточена вся бизнес-логика, правила, ограничения и модели, описывающие реальные процессы предметной области. Этот уровень оперирует понятийным аппаратом, согласованным с экспертами, что обеспечивается принципом единого языка между разработчиками и бизнесом.

Модели домена – сущности, объекты – значения, агрегаты, доменные события и интерфейсы репозиториев. Несмотря на то, что доменный слой контролирует и использует текущее состояние бизнес-модели, он делегирует технические аспекты, такие как сохранение данных или интеграция с внешними API, на нижележащие уровни.

К примеру, в образовательной системе на этом уровне будут представлены такие сущности (домены), как курс, материал или студент. Также могут быть объекты – значения, например, семестр, для которого дополнительно определен механизм валидации, не позволяющий использовать недопустимые значения.

Таким образом, уровень предметной области не просто хранит данные, а определяет смысл этих данных и правила их трансформации в рамках бизнес-процессов.

Инфраструктурный слой обеспечивает техническую поддержку всех вышестоящих уровней. Он реализует детали взаимодействия с внешним миром: хранение и извлечение данных, взаимодействие с другими системами, обмен сообщениями, интеграцию с фреймворками и библиотеками.

К задачам инфраструктурного уровня в архитектуре Domain-Driven Design относятся такие аспекты, как реализация репозиториев для абстрагирования от способа хранения данных и предоставления единого интерфейса для взаимодействия с ними, а также организация отправки электронных писем в рамках работы приложений. Кроме того, сюда входит управление подключением к базам данных для хранения и извлечения информации.

Также инфраструктурный уровень отвечает за взаимодействие с файловой системой, позволяя корректно управлять доступом к файлам и обеспечивать необходимые операции с ними. Наконец, важной составляющей этого уровня является журналирование (логирование) и мониторинг, которые обеспечивают контроль за состоянием системы, помогают оперативно реагировать на ошибки и анализировать производительность приложения.

Следует подчеркнуть, что зависимости между уровнями архитектуры направлены сверху вниз: инфраструктурный уровень зависит от доменного, но не наоборот. Это обеспечивает возможность замены технических реализаций без затрагивания бизнес-логики.

### 2.1.2. Архитектура клиентской части

Современные фронтенд-приложения (клиентские части) отличаются высокой степенью сложности и разнообразием задач, выходящих далеко за рамки простого отображения HTML-контента. В условиях, когда интерфейсная часть приложения включает десятки или даже сотни независимых компонентов, становится необходимым формализованный подход к организации кода. Эффективная архитектура фронтенда способствует масштабируемости, упрощает тестирование, облегчает командную разработку и повышает читаемость проекта.

React – библиотека JavaScript, ориентированная на построение компонентных пользовательских интерфейсов [6]. Её основной концепцией является декларативное описание UI с помощью компонентов, обладающих собственным состоянием и логикой жизненного цикла. Именно компонентный подход, заложенный в основу React, формирует предпосылки для модульной архитектуры: приложение представляется как дерево вложенных компонентов, каждый из которых может быть спроектирован, протестирован и переиспользован независимо.

Благодаря этой парадигме разработчики получают возможность изолировать поведение, делегировать ответственность и формировать слои абстракции, отражающие как технические, так и бизнес-аспекты интерфейса. Однако, по мере роста проекта, простая иерархия компонентов перестаёт быть достаточной, и возникает необходимость в архитектурной модели, регламентирующей правила проектирования и организации компонентов.

Одной из наиболее популярных и широко применяемых методологий организации фронтенд-архитектуры является Atomic Design, предложенный Бредом Фростом [7]. Данный подход предлагает структурировать компоненты интерфейса по степени их абстракции и переиспользуемости, используя аналогию с химическими элементами. Методология включает пять уровней:

Атомы (Atoms) – базовые элементы интерфейса, не подлежащие дальнейшей декомпозиции. Это, например, Button, Input, Label, Checkbox, стилизованные иконки и т.п. Они не зависят от бизнес-логики и переиспользуются в более сложных компонентах.

Молекулы (Molecules) – комбинации атомов, объединённые для выполнения конкретной задачи. Пример: форма ввода, включающая поле Input и кнопку Submit.

Организмы (Organisms) – более сложные структуры, включающие в себя несколько молекул и атомов. Пример: навигационная панель с логотипом, списком ссылок и кнопкой входа в систему.

Шаблоны (Templates) – макеты страниц, описывающие структуру расположения компонентов. Они задают layout, но не содержат конкретных данных.

Страницы (Pages) – конкретные реализации шаблонов с привязанными данными и логикой взаимодействия. Это компоненты, связанные с маршрутизацией, инициализацией данных и состоянием.

Такой подход позволяет чётко разграничить переиспользуемые и специфические компоненты, что обеспечивает баланс между гибкостью и структурностью. При этом слои архитектуры не жёстко навязаны, а развиваются эволюционно – в зависимости от сложности проекта.

### 2.1.3. Клиент-серверное взаимодействие

Классическая модель клиент-серверного взаимодействия в некоторых случаях имеет практические ограничения. По мере роста сложности систем и увеличения числа пользователей появляются слабые места прямого взаимодействия клиента с сервером приложения. Среди них – невозможность централизованно управлять маршрутизацией запросов, ограниченная гибкость в масштабировании, затруднённое обеспечение безопасности.

Следовательно, необходим промежуточный уровень между клиентом и внутренними сервисами системы компонента, способного централизованно обрабатывать входящий трафик, управлять им, фильтровать, преобразовывать и направлять в нужную точку внутренней архитектуры.

Обратный прокси-сервер – это сервер, который принимает запросы от клиентов и перенаправляет их на один или несколько серверов приложений. Он выступает в роли посредника между клиентами и серверами, обеспечивая дополнительные функции, такие как балансировка нагрузки, кэширование, сжатие и обеспечение безопасности. В данном случае роль обратного прокси-сервера выполняет Nginx [8].

В типичном сценарии взаимодействия пользователь инициирует HTTP-запрос, который в первую очередь обрабатывается Nginx. Он, в зависимости от настроек, может направить этот запрос на один из внутренних серверов приложений, передать его статической файловой системе, преобразовать в другой формат, либо отклонить с соответствующим кодом ответа. Благодаря этому Nginx выступает как точка входа в систему, обеспечивающая её логическую и физическую изоляцию от внешней среды.

Особенность Nginx заключается в том, что он работает на низком уровне с протоколами и соединениями, тем самым разгружая основное приложение от необходимости решать задачи, напрямую не связанные с его бизнес-логикой. Например, при использовании TLS (ранее SSL) шифрование и расшифровка трафика могут быть целиком возложены на Nginx, что упрощает внутреннюю архитектуру и снижает нагрузку на серверные компоненты.

Кэширование – ещё один важный аспект. Nginx может кэшировать ответы от серверов приложений, сохраняя их на диске или в памяти. Это особенно актуально для статического или редко изменяющегося контента, такого как изображения, стили или общие данные. Кэширование снижает количество повторных обращений к серверу, ускоряет отклик системы и уменьшает нагрузку на внутренние ресурсы.

## 2.2. Проектирование базы данных

### 2.2.1. ER-модель базы данных

Для разработки ER-модели первым этапом необходимо выделить все имеющиеся сущности проекта, а также описать связи между ними.

Курс (Course) – сильная сущность, представляющая изучаемые курсы в университете. Содержит атрибуты: код курса (courseId), название курса (name), краткое описание курсы (description), семестр курса (semester), а также имя преподавателя, которые ведет данный курс (teacher) и количество часов, посвященное данному курсу (hours). Сущность идентифицируется по коду курса.

Материал (Material) – слабая сущность, которая представляет собой различные материалы для конкретного курса. Содержит атрибуты: код материала (materialId), тип материала (type), например, лекции. Название материала (name) и его краткое описание (content), а также номер материала (number). Сущность идентифицируется по коду материала.

Файл (File) – также слабая сущность, которая является файлом, прикрепленным к конкретному материалу. Содержит атрибуты: код файла (fileId), название файла (name), описание файла (description) и ссылка на местоположение файла (url). Сущность идентифицируется по коду файла.

Пользователь (User) – сильная сущность, представляющая собой пользователей (студенты или преподаватели), которые будут пользоваться данным сервисом. Содержит атрибуты: код пользователя (userId), имя пользователя (username), электронная почта (email), хэш пароля (hashedPassword), а также флаг (isTeacher), которой в следствии будет наделять особыми правами преподавателей. Сущность идентифицируется по коду пользователя.

Сессия (Session) – слабая сущность, которая представляет собой активные сессии авторизации пользователя. Ее атрибутами являются: код сессии (sessionId) и время, в которое сессия истекает (expiredAt). Сущность идентифицируется по коду сессии

Связь между сущностями Course и Material показывает зависимость материалов от курсов. Каждый материал обязательно принадлежит одному курсу, в то время как, курс может содержать множество материалов (или не содержать их вовсе).

Связь между сущностями Material и File также является связью “Один ко многим”. Один материал можно содержать в себе множество файлов (или не содержать совсем). Каждый файл прикрепляется к одному материалу. Файл не может существовать в системе без привязки к материалу.

Связь между Course и User является связью “многие ко многим”. Как у курса может быть несколько пользователей (студенты, которые осваивают данный курс), так и у пользователя может быть несколько курсов (студент изучает несколько курсов параллельно).

Связь между User и Session представляет собой отношение между пользователями и их сессиями входа. Один пользователь может иметь несколько активных или завершенных сессий, Каждая сессия должна быть привязана к какому-то конкретному пользователю. Данная связь является “один ко многим”. На рисунке 2 представлена смеха разработанной модели.

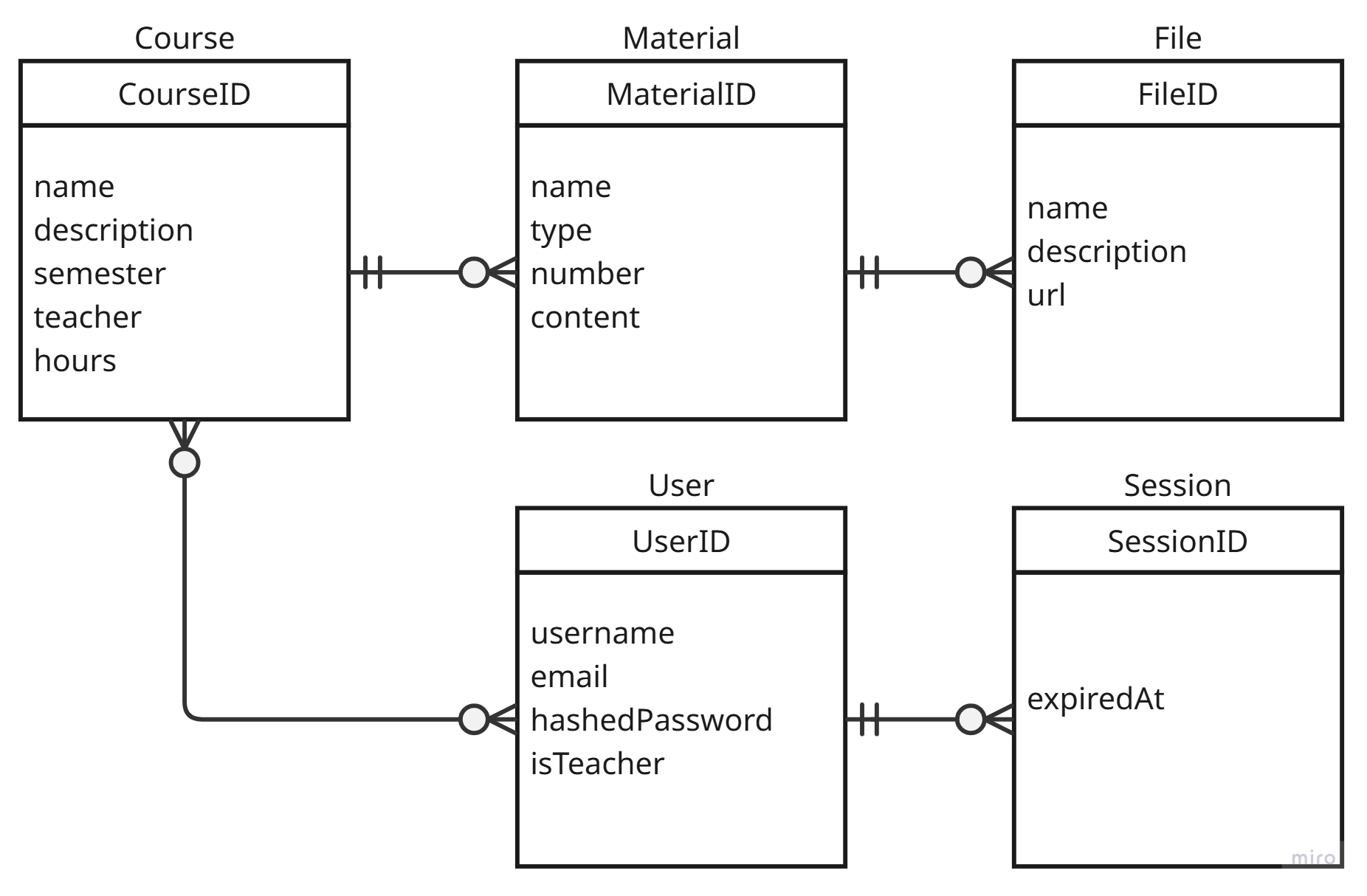


Рисунок 2 – ER-модель базы данных

### 2.2.2. Преобразование ER-модели в реляционную модель

Курс (Course). В качестве первичного ключа – уникальный числовой идентификатор (сourseId) – суррогатный ключ. Содержит поля: название курса (name), краткое описание курса (description), семестр курса (semester), имя преподавателя (teacher), количество часов (hours).

Материал (Material). В качестве первичного ключа – уникальный числовой идентификатор (materialId) – суррогатный ключ. Содержит поля: тип материала (type), название материала (name), краткое описание (content), номер материала (number). Также содержит внешний ключ курс (courseId), ссылающийся на отношение Course.

Файл (File). В качестве первичного ключа – уникальный идентификатор (fileId), созданный по стандарту UUID версии 4. Содержит поля: название файла (name), описание файла (description), ссылка на местоположение файла (url). Также содержит внешний ключ материал (materiald), ссылающийся на отношение Material.

Пользователь (User). В качестве первичного ключа – уникальный идентификатор (userId). Содержит поля: имя пользователя (username), электронная почта (email), хэш пароля (hashedPassword), флаг для определения типа пользователя (isTeacher).

Сессия (Session). В качестве первичного ключа – уникальный идентификатор (sessionId), который генерируется с помощью хэширования случайных байтов по алгоритму SHA 256. Содержит поля: время истечения срока действия (expiredAt). Также содержит внешний ключ пользователь (userId), ссылающийся на отношениеUser.

Избранные (Featured) – промежуточное отношение для обеспечения связи между пользователями и курсами, обеспечивающей функционал избранных курсов. В качестве первичного ключа – составной ключ из внешний ключей отношений User и Course – userId и courseId.

Курс связан с материалом. Материал связан с файлом. Пользователь связан с сессией. Курс связан с избранными курсами. Пользователь связан и избранными курсами. У всех связей тип – Mandatory-to-Optional. One-to-Many.

На рисунке 3 представлена схематическая реализация реляционной модели описанной базы данных, которая позволяет реализовать необходимые возможности.

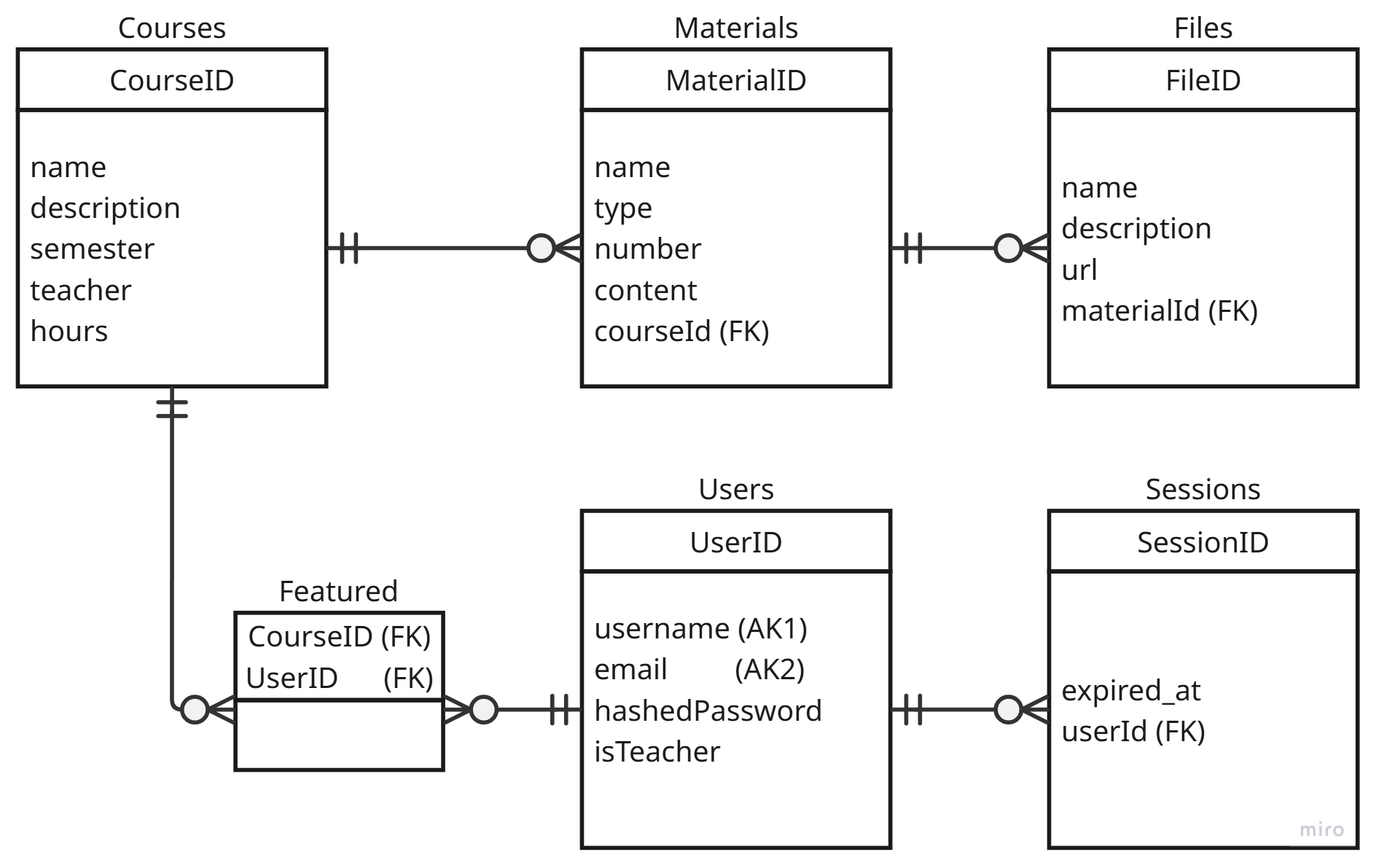


Рисунок 3 - Реляционная модель базы данных

Теперь на основе описания связей определим их кардинальности (таблица 1) и действия для ограничения минимальной кардинальности для каждой связи (таблицы 2 – 6).

Таблица 1 – Кардинальность связей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Связь | | Кардинальность | | |
| Родитель | Потомок | Тип | Макс. | Мин. |
| Course | Material | Идентифицирующая | 1-N | M-O |
| Material | File | Идентифицирующая | 1-N | M-O |
| User | Session | Идентифицирующая | 1-N | M-O |
| Course | Featured | Идентифицирующая | 1-N | M-O |
| User | Featured | Идентифицирующая | 1-N | M-O |

Таблица 2 – Ограничение связи Course-Material

|  |  |  |
| --- | --- | --- |
|  | Действие над Course | Действие над Material |
| Вставка | - | Подбор записи |
| Изменение ключа | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 3 – Ограничение связи Material-File

|  |  |  |
| --- | --- | --- |
|  | Действие над Material | Действие над File |
| Вставка | - | Подбор записи |
| Изменение ключа | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 4 – Ограничение связи User-Session

|  |  |  |
| --- | --- | --- |
|  | Действие над User | Действие над Session |
| Вставка | - | Подбор записи |
| Изменение ключа | Запрет | Запрет |
| Удаление | Запрет | - |

Таблица 5 – Ограничение связи Course-Featured

|  |  |  |
| --- | --- | --- |
|  | Действие над Course | Действие над Featured |
| Вставка | - | Подбор записи |
| Изменение ключа | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 6 – Ограничение связи User-Featured

|  |  |  |
| --- | --- | --- |
|  | Действие над User | Действие над Featured |
| Вставка | - | Подбор записи |
| Изменение ключа | Запрет | Запрет |
| Удаление | Запрет | - |

Под конец остается только описать отношения в базе данных (таблицы 7-13), а именно определить типы полей, типы ключей, допустимость неопределенных значений, а также какую-либо дополнительную информацию.

Таблица 7 – Отношение Course

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| courseId | Int | Первичный | Нет | Суррогатный |
| name | Varchar(35) |  | Нет |  |
| description | Text |  | Да | По умолчанию пустая строка |
| semester | Int |  | Нет | От 1 до 8 |
| teacher | Varchar(35) |  | Да |  |
| hours | Int |  | Да |  |

Таблица 8 – Отношение Material

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| materialId | Int | Первичный | Нет | Суррогатный |
| courseId | Int |  | Нет | FK |
| type | Varchar(35) |  | Нет | «lectures», «seminars», «labs» |
| name | Varchar(35) |  | Да | По умолчанию - «default\_name» |
| content | Text |  | Да |  |
| number | Int |  | Да |  |

Таблица 9 – Отношение File

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| fileId | Varchar(255) | Первичный | Нет | UUIDv4 |
| materialId | Int |  | Нет | FK |
| name | Varchar(255) |  | Нет |  |
| description | Varchar(255) |  | Да |  |
| url | Varchar(255) |  | Нет |  |

Таблица 10 – Отношение User

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| userId | Int | Первичный | Нет | Суррогатный |
| username | Varchar(255) | Кандидат | Нет | AK1 |
| emails | Varchar(255) | Кандидат | Нет | AK2 |
| hashedPassword | Varchar(255) |  | Нет | Хэш |
| isTeacher | Bool |  | Нет | По умолчанию False |

Таблица 11 – Отношение Session

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| sessionId | Varchar(255) | Первичный | Нет | SHA-256 |
| userId | Int |  | Нет | FK |
| expiredAt | Datetime |  | Нет |  |

Таблица 12 – Отношение Featured

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Столбец | Тип | Ключ | Неопр. знач. | Дополнительно |
| userId | Int | Первичный | Нет | FK |
| courseId | Int | Первичный | Нет | FK |

## 2.3. Взаимодействие с большими языковыми моделями

Большие языковые модели (LLM) представляют собой мощный инструмент для обработки и генерации текста. Однако их стандартное применение имеет фундаментальное ограничение: их знания статичны и ограничены данными, на которых они были обучены. Это приводит к двум основным проблемам: во-первых, модели не осведомлены о событиях или данных, появившихся после их обучения; во-вторых, они не имеют доступа к частной, узкоспециализированной информации. Попытки модели ответить на вопросы в таких областях без дополнительных данных могут приводить к так называемым "галлюцинациям" — генерации правдоподобных, но фактически неверных или вымышленных ответов.

Для преодоления этих ограничений в современной инженерии систем с использованием ИИ применяется архитектурный паттерн RAG, или "генерация, дополненная извлечением данных". RAG – это гибридный подход, который объединяет мощные генеративные и логические способности предобученной языковой модели с возможностью динамического доступа к базе данных. Вместо того чтобы полагаться исключительно на информацию, находящуюся в параметрах модели, система сначала извлекает релевантные данные из внешнего источника, а затем использует их в качестве контекста для генерации ответа.

Процесс обработки запроса в рамках RAG можно разделить на следующие этапы:

1. Распознавание намерения и формирование поискового запроса. На первом этапе система анализирует входящий запрос пользователя для определения его основного намерения. Целью является понять, какую именно информацию ищет пользователь. На основе этого анализа система формирует формализованный поисковый запрос к внешней базе знаний.
2. Извлечение информации. Сформированный поисковый запрос выполняется над внешней базой знаний. В качестве такой базы может выступать реляционная база данных. В результате этого этапа система получает набор текстовых фрагментов, документов или записей, которые с высокой вероятностью являются релевантными для ответа на исходный вопрос пользователя.
3. Контекстное обогащение и конструирование запроса.
4. Генерация ответа. Финальный, обогащенный запрос передается большой языковой модели. Модель, используя свои внутренние способности к рассуждению и обработке языка, синтезирует целостный ответ, основываясь в том числе на предоставленном контексте, а не только на своих общих знаниях.

Нейросетевые модели можно задействовать двумя способами: разместить на собственном сервере или воспользоваться сторонним сервисом, предоставляющим доступ к нейросетевой модели с помощью интерфейса API.

Первый способ довольно затратный и требует специальных высокопроизводительных серверов. Несмотря на то, что есть некоторые сильно уменьшенные большие языковые модели, которые могут работать на обычных компьютерах, их способности сильно ограничены. Поэтому использование собственных нейросетевых моделей в рамках данной дипломной работы нецелесообразно.

Второй способ предполагает использование нейросетевого вывода, предоставляемого тем или иным сторонним сервисом бесплатно или за определённую плату. Существует широкий спектр GPT‑подобных моделей, предоставляемых через API ведущими компаниями и исследовательскими группами, которые можно условно разделить на несколько категорий.

Во‑первых, OpenAI предлагает семейство GPT‑моделей, доступных через Chat Completions API. Сейчас в распоряжении разработчиков находятся модели GPT‑4.1, GPT‑4.1 mini и GPT‑4.1 nano, запущенные в мае–июне 2025 года. Они характеризуются повышенной производительностью при кодировании, улучшенными способностями к обработке длинного контекста (до 1 млн токенов), а также сниженной стоимостью и задержками по сравнению с предыдущими версиями, включая GPT‑4o и GPT‑4.5. Помимо этого, OpenAI продолжает поддерживать модели с функцией рассуждений (оба варианта o‑семейства: o3, o4‑mini) и предоставляет доступ к специализированным моделям для генерации изображений, таким как gpt‑image‑1.

Во‑вторых, Microsoft посредством Azure AI предлагает GPT‑3.5 Turbo и GPT‑3.5 Turbo Instruct через Azure OpenAI Service, а также интегрирует GPT‑4о в свой сервис Copilot, основанный на «Prometheus» ‑модели, которая включает GPT‑4, GPT‑4 Turbo и GPT‑4o.

Третье направление представляют европейские и российские поставщики. Французский стартап Mistral предлагает открытые LLM‑модели Magistral Small и коммерческие Magistral Medium через Hugging Face и API, активно продвигая их в качестве конкурентов OpenAI и Google.

Немецкая компания Aleph Alpha развивает семейство Luminous и предоставляет API‑доступ, ориентированный на Европу и отвечающий требованиям конфиденциальности и прозрачности.

Российская компания Яндекс распространяет YandexGPT через Yandex Cloud, предлагая как асинхронные, так и реальные API‑механизмы для бизнеса.

Кроме того, IBM предоставляет платформу Watsonx, включая собственные LLM под названием Granite и другие модели с акцентом на корпоративную безопасность, настройку и приватность данных.

Канадская компания Cohere предлагает API для генеративных задач, классификации, извлечения данных и модерации, поддерживая развертывание через облачные платформы Amazon и Google, а также выпустив Aya Vision — визуально ориентированную модель.

Стоит также упомянуть китайские инициативы с открытым исходным кодом, такие как dots.llm1 от Rednote и Qwen‑серия от Alibaba, доступные через площадку Hugging Face.

# 3. Реализация системы

## 3.1. Выбор стека технологий

Клиентская часть приложения реализована как современное одностраничное приложение. В качестве основного фреймворка используется React в связке с TypeScript [9]. Такой выбор обеспечивает компонентный подход к построению интерфейса и строгую типизацию, что повышает надежность и облегчает поддержку кода. Для ускорения процесса разработки и оптимизации сборки применяется инструментарий Vite. Пользовательский интерфейс создан с помощью комбинации двух популярных библиотек: Ant Design предоставляет богатый набор готовых компонентов, таких как формы, таблицы и модальные окна, в то время как React Bootstrap используется для создания адаптивной сетки и структуры страниц. Навигация в приложении, не требующая перезагрузки страницы, реализована с помощью React Router [10]. Для управления общим состоянием, например, данными о текущем пользователе, используется легковесная библиотека Zustand. Взаимодействие с серверной частью осуществляется через HTTP-клиент Axios.

Серверная часть написана на языке Python [11] и представляет собой высокопроизводительное асинхронное приложение. В качестве основного фреймворка был выбран FastAPI. Взаимодействие с базой данных построено на основе SQLAlchemy [12] – ORM-инструмента, который позволяет работать с таблицами как с объектами Python. Управление версиями схемы базы данных (миграциями) осуществляется с помощью Alembic.

В качестве СУБД был выбран PostgreSQL – объектно-реляционная система управления базами данных, которая обеспечивает надежное хранение информации.

## 3.2. Схема программного обеспечения

Программное обеспечение проекта построено на основе архитектуры с четким разделением ответственности между клиентской частью (фронтенд), серверной частью (бэкенд) и инфраструктурными компонентами. Взаимодействие между пользователем и системой, а также между основными компонентами оптимизировано и защищено посредством обратного прокси-сервера Nginx.

На рисунке 4 представлена общая схема программного обеспечения системы, иллюстрирующая основные компоненты и их взаимосвязи.

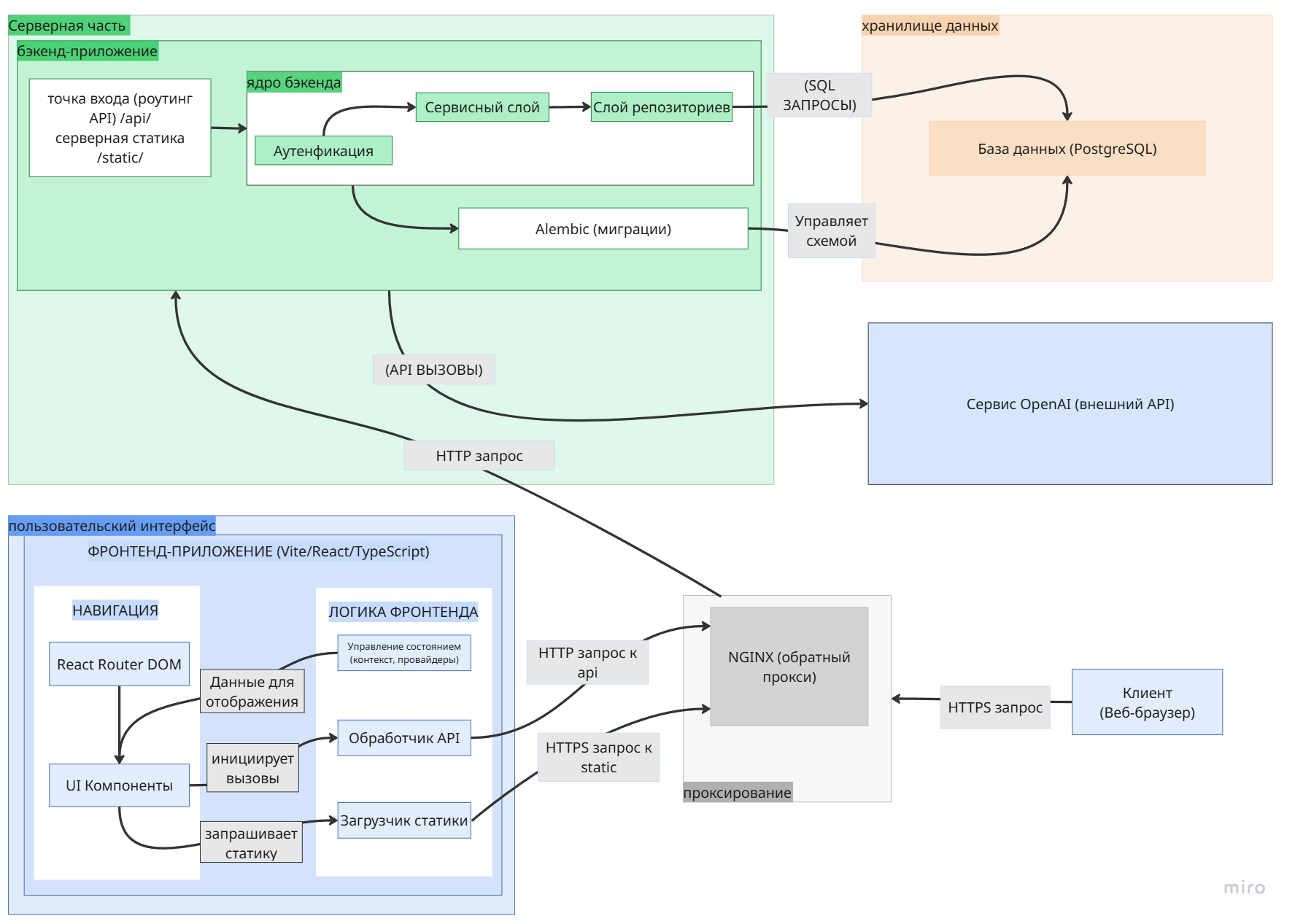


Рисунок 4 – Схема ПO

Пользовательский интерфейс реализован с помощью технологий Vite, React и TypeScript, и полностью отвечает за взаимодействие с пользователем. Навигация между различными экранами реализуется посредством React Router DOM. Логика фронтенда объединяет несколько важных функций: управление состоянием приложения, которое осуществляется через провайдеры и контексты, для коммуникации с серверной частью предусмотрен обработчик API, который делает запросы и получает ответы.

Серверная часть обрабатывает основную бизнес-логику и запросы на получение или изменение данных. Бэкенд имеет точку входа, которая обеспечивает маршрутизацию для API-запросов. Серверная часть приложения реализована по слоям, сохраняя концепцию предметно-ориентированного проектирования архитектуры.

Хранение данных приложения осуществляется в реляционной базе данных PostgreSQL. Серверная часть приложения взаимодействует с этой базой данных через репозитории (инфраструктурный уровень), которые осуществляют SQL-запросы для чтения и запись информации.

Система также интегрирована с внешним сервисом OpenAI. Бэкенд-приложение выполняет API-вызовы, представляющие собой HTTP-запросы, к этому внешнему API, чтобы использовать предоставляемые им возможности и функции.

## 3.3. Серверная часть

### 3.3.1. Инфраструктурный уровень

Для удобного и безопасного хранения параметров подключения к базе данных, а также дополнительных настроек, используется библиотека pydantic-settings. Она позволяет загружать конфигурацию из различных источников, в том числе из переменных окружения и env файлов. Параметры подключения к базе данных PostgreSQL определены в классе DbSettings (см. Листинг 1), наследуемом от BaseSettings, в виде соответствующих полей класса. Типы этих полей библиотека pydantic проверяет автоматически. Дополнительно определён метод get\_db\_url, который конструирует URL-строку подключения для SQLAlchemy.

from pydantic\_settings import BaseSettings, SettingsConfigDict

class DbSettings(BaseSettings):

ECHO: bool

USER: str

PASSWORD: str

HOST: str

PORT: int

NAME: str

model\_config = SettingsConfigDict(

env\_file="db.env", env\_file\_encoding="utf-8")

def get\_db\_url(self) -> str:

return f"postgresql+asyncpg://{self.USER}:{self.PASSWORD}@{self.HOST}:{self.PORT}/{self.NAME}"

# ... другие классы настроек ...

class Settings(BaseSettings):

db: DbSettings = DbSettings() # Экземпляр настроек БД

# ... другие настройки приложения ...

settings = Settings()

Листинг 1 – Конфигурация подключения к базе данных

Класс DbSettings наследуется от BaseSettings и определяет поля, соответствующие необходимым параметрам подключения. Pydantic автоматически проверяет типы этих параметров при загрузке. Метод get\_db\_url конструирует URL-строку подключения для SQLAlchemy.

После того как параметры подключения загружены, они используются для создания движка SQLAlchemy, который управляет пулом соединений с базой данных и “фабрики сессий”, которая создает сессии для взаимодействия с базой данных. Данная логика реализована в классе DatabaseHelper и продемонстрирована в листинге 2.

class DatabaseHelper:

def \_\_init\_\_(self, url: str, echo: bool = False):

self.engine = create\_async\_engine(

url=url,

echo=echo

)

self.session\_factory: async\_sessionmaker[AsyncSession] = async\_sessionmaker(

bind=self.engine,

expire\_on\_commit=False,

autoflush=False,

autocommit=False,

)

async def session\_getter(self) -> AsyncGenerator[AsyncSession, None]:

async with self.session\_factory() as session:

yield session

db\_helper = DatabaseHelper(

url=settings.db.get\_db\_url(),

echo=settings.db.ECHO

)

Листинг 2 – Реализация класса DatabaseHelper

Метод create\_async\_engine создает асинхронный движок, который является центральной точкой доступа к базе данных. async\_sessionmaker создает фабрику, которая будет производить экземпляры AsyncSession. Сессия SQLAlchemy – это основной интерфейс для выполнения запросов к базе данных. Она отслеживает изменения в объектах и координирует транзакции. Метод session\_getter предоставляет асинхронную сессию для каждого HTTP-запроса, который требует взаимодействия с БД, и гарантирует, что сессия будет правильно закрыта после завершения запроса.

Как было упомянуто выше, SQLAlchemy является ORM-фреймворком, где ORM обозначает объектно-реляционное отображение. Суть этой технологии заключается в установлении моста между объектно-ориентированной парадигмой программирования, используемой в приложении (в нашем случае, на языке Python) и реляционной моделью данных, используемой в базе данных

Вместо того, чтобы писать sql-запросы для каждой операции с базой данных, можно определить структуру таблиц базы данных в виде Python-классов. Эти классы называются ORM моделями, каждая из которых соответствует одной таблице в базе данных, атрибуты классы соответствуют столбцам таблицы, а экземпляры этого класса представляют собой строки в таблице.

В данном проекте ORM модели располагаются в директории orm\_models. Рассмотрим пример ORM модели для сущности “Курс”, представленный в листинге 3.

class CourseModel(Base):

\_\_tablename\_\_ = "courses"

id: Mapped[int] = mapped\_column(

Integer, primary\_key=True, index=True, autoincrement=True)

name: Mapped[str] = mapped\_column(String(255), nullable=False)

description: Mapped[str] = mapped\_column(Text, nullable=True)

semester: Mapped[int] = mapped\_column(Integer, nullable=False)

teacher: Mapped[str] = mapped\_column(String(255), nullable=True)

hours: Mapped[int] = mapped\_column(Integer, nullable=True)

materials: Mapped[list["MaterialModel"]] = relationship(

back\_populates="course", cascade="all, delete-orphan"

)

featured\_by: Mapped[list["UserModel"]] = relationship(

secondary="user\_featured\_courses",

back\_populates="featured\_courses",

lazy="noload"

)

featured\_courses\_association: Mapped[list["UserFeaturedCourseModel"]] = relationship(

back\_populates="course",

cascade="all, delete-orphan",

lazy="noload"

)

Листинг 3 – ORM модель для сущности “Курс”

CourseModel – Python-класс, который будет представлять таблицу courses в базе данных. Атрибуты класса соответствуют столбцам таблицы courses, а также с помощью mapped\_column можно указать их тип данных, а также свойства (первичный ключ, возможность быть NULL, индексация, автоинкремент и т.д.).

SQLAlchemy представляет отношения в реляционных БД с помощью механизма relationship, который позволяет определить эти связи на уровне ORM моделей. Это означает, что можно обращаться к связанным данным так, как если бы они были обычными атрибутами Python-объектов, а SQLAlchemy позаботится о генерации необходимых SQL-запросов (включая JOIN'ы) для их извлечения. В листинге № 3 определены два типа связей: один-ко-многим и многие-ко-многим, рассмотрим их подробнее.

Связь «Один-ко-многим». Эта связь означает, что один экземпляр CourseModel может быть связан несколькими экземплярами MaterialModel. back\_populates это очень важный параметр для определения двунаправленной связи. Он указывает, что в модели MaterialModel также должно быть определено отношение, которое будет ссылаться обратно на CourseModel. Это позволяет SQLAlchemy синхронизировать обе стороны отношения. Также задается параметр, который управляет поведением при операциях с родительским объектом

Связь «Многие-ко-многим». Эта связь означает, что один курс может быть «избранным» многими пользователями, и один пользователь может «избрать» много курсов. Связи «многие-ко-многим» в реляционных базах данных реализуются через промежуточную (ассоциативную) таблицу, в нашем случае таблица user\_featured\_courses выступает таковой. Эта таблица содержит как два столбца с внешними ключами, ссылающимися на первичные ключи таблиц для курсов и пользователей.

По мере развития приложения его требования к структуре базе данных могут меняться. Могут потребоваться новые таблицы, столбцы в существующих таблицах, изменение типов данных, добавление или удаление или удаление индексов и ограничений. Управление этими изменениями вручную может быть сложным, подверженным ошибкам.

Для решения этой проблемы используется Alembic – инструмент для управления миграциями схемы базы данных. Каждое изменение схемы оформляется в виде отдельной миграции, которая имеет уникальный идентификатор и может быть применена или отменена.

Рассмотрим пример работы с миграциями на уже известной нам сущности “Курс”. При создании (или изменении) ORM модели (например, CourseModel) необходимо выполнить команду Alembic для автоматической генерации скрипта миграции (см. листинг 4). Alembic сравнивает модели с текущей схемой базы данных и создает файл alembic/revision/. Пример миграции представлен в листинге 5.

alembic revision -m "add\_new\_field\_to\_courses" --autogenerate

Листинг 4 – Команда Alembic для генерации миграции

"""add column hours to table Courses

Revision ID: 58901a5f6d2a

Revises: e9374f83547d

Create Date: 2025-05-07 16:31:53.301580"""

from typing import Sequence, Union

from alembic import op

import sqlalchemy as sa

revision: str = '58901a5f6d2a'

down\_revision: Union[str, None] = 'e9374f83547d'

branch\_labels: Union[str, Sequence[str], None] = None

depends\_on: Union[str, Sequence[str], None] = None

def upgrade() -> None:

op.add\_column('courses', sa.Column('hours', sa.Integer(), nullable=True))

def downgrade() -> None:

op.drop\_column('courses', 'hours')

Листинг 5 – Пример миграции для создания нового столбца в таблице courses

Функция upgrade описывает, как применить изменение, а downgrade – как его отменить. Для применения миграции и обновления схемы базы данных до последней версии выполняется команда, представленная в листинге 6. Эта команда применит все новые миграции, которых еще нет в базе данных.

alembic upgrade head

Листинг 6 – Команда для применения миграции

Для организации чистого и, что самое главное, поддерживаемого кода при взаимодействии с базой данных в проекте применяется паттерн “Репозиторий”. Этот паттерн служит для внедрения логики доступа к данным, представляя своего рода коллекцию объектов, с которой может работать остальная часть приложения, не задумываясь о деталях хранения и извлечения этих объектов из базы данных.

Рассмотрим пример взаимодействия с базой данных в репозитории для сущности “Курс”, представленный в листинге 7

class CourseRepository(ICourseRepository):

def \_\_init\_\_(self, session: AsyncSession):

self.session = session

async def get\_courses(self) -> list[Course]:

stmt = select(CourseModel).order\_by(CourseModel.id)

result: Result = await self.session.execute(stmt)

courses = result.scalars().all()

return [Course.model\_validate(course) for course in courses]

async def create\_course(self, course: Course) -> Course:

new\_course = CourseModel(\*\*course.model\_dump())

self.session.add(new\_course)

await self.session.commit()

return Course.model\_validate(new\_course)

async def get\_course\_by\_id(self, course\_id: int) -> Course:

stmt = select(CourseModel).where(CourseModel.id == course\_id)

result: Result = await self.session.execute(stmt)

course = result.scalar\_one\_or\_none()

return Course.model\_validate(course)

Листинг 7 – Репозиторий курсов

В приведенном фрагменте CourseRepository получает асинхронную сессию SQLAlchemy (AsyncSession) через конструктор. Методы репозитория используют эту сессию для выполнения асинхронных запросов к базе данных. Важной частью работы репозитория является также преобразование данных между ORM-моделями (например, CourseModel), которые используются для взаимодействия с SQLAlchemy, и доменными моделями (например, Pydantic-модель Course), которые используются в бизнес-логике и API слоях приложения. Для взаимодействия с базой данной используются функции SQLAlchemy, для построения запросов, выполнения запросов, извлечения данных из результата, различные модификации данных, фиксация изменений.

### 3.3.2. Доменный уровень

В соответствии с принципами предметно-ориентированного проектирования, доменный уровень является основой всей серверной архитектуры. Именно здесь концентрируется вся бизнес-логика, правила и процессы приложения. Этот слой отвечает за представление концептуальной модели предметной области в коде, обеспечивая точное отражение реальных сущностей в приложении. Структура доменного слоя состоит из доменных моделей и интерфейсов репозиториев.

Модели реализованы с использованием библиотеки Pydantic, что обеспечивает валидацию данных и удобную сериализацию и десериализацию. Они содержат атрибуты, отражающие свойства сущностей, и могут включать методы, инкапсулирующие доменную логику (бизнес-правила), специфичную для данной сущности. В отличие от ORM-моделей, доменные модели фокусируются на поведении и правилах предметной области. Всего в проекте пять моделей: модель курсов (Course), пользователей (User), материалов (Material, файлов (MaterialFile) и сессий (Session).

Интерфейсы указывают, какие операции по работе с данными должны быть доступны для каждой сущности (например, «найти курс по уникальному идентификатору», «сохранить пользователя»). Конкретные реализации этих интерфейсов находятся в инфраструктурном слое. Пример сущности “Курс” (Course) продемонстрирован в листинге 8.

class Course(BaseModel):

name: str

semester: int

id: Optional[int] = None

description: Optional[str] = None

teacher: Optional[str] = None

hours: Optional[int] = None

model\_config = ConfigDict(from\_attributes=True)

Листинг 8 – Сущность “Курс” в предметном слое

### 3.3.3. Прикладной уровень

Прикладной уровень отвечает за реализацию конкретных сценариев использования системы. Он управляет взаимодействием между пользовательскими запросами, поступающими с уровня представления, и бизнес-логикой в предметном слое. В структуре проекта этот слой представлен сервисами, находящимися в директории services.

Основными компонентами этого слоя являются классы, такие как UsersService, CoursesService и MaterialService. Каждый сервис группирует операции, относящиеся к определенной функциональной области системы. Рассмотрим, например, сервис CoursesServices, представленный в листинге 9.

class CoursesService:

def \_\_init\_\_(self, course\_repo: ICourseRepository):

self.course\_repo = course\_repo

async def get\_courses(self) -> List[Course]:

return await self.course\_repo.get\_courses()

async def create\_course(self, name: str, description: Optional[str], semester: int, teacher: Optional[str], hours: Optional[int]) -> Course:

new\_course\_domain\_obj = Course(

name=name,

description=description,

semester=semester,

teacher=teacher,

hours=hours

)

created\_course = await self.course\_repo.create\_course(new\_course\_domain\_obj)

return created\_course

Листинг 9 – Реализация сервиса курсов

CoursesService зависит от интерфейса ICourseRepository. При инициализации сервиса ему будет передан конкретный экземпляр репозитория, реализующий данный интерфейс (например, CourseRepository из инфраструктурного слоя). Это обеспечивает слабую связность: CoursesService зависит от абстракции, а не от конкретной реализации.

Методы сервисов представляют собой реализацию конкретных сценариев использования. Например, метод для создания нового курса выполняет следующие действия:

1. Принимает параметры, необходимые для создания курса.
2. Создает экземпляр предметной (доменной) модели Course. На этом этапе происходит валидация данных с использованием правил, определенных в Pydantic-модели Course.
3. Вызывает метод для создания курса в репозитории, передавая ему созданный доменный объект. Репозиторий отвечает за непосредственное сохранение данных в базу.
4. Возвращает созданный и сохраненный доменный объект Course.

Важно также отметить, что сервисы на прикладном уровне архитектуры DDD не включают в себя непосредственную бизнес-логику, а предназначены исключительно для координации процессов между различными компонентами системы.

В частности, сервис пользователей отвечает за управление жизненным циклом пользователей и обеспечивает их аутентификацию. Помимо этого, он управляет процессами, связанными с добавлением и удалением избранных курсов, так как эти операции непосредственно привязаны к профилю пользователя. В свою очередь, сервис курсов предоставляет функциональность для создания и получения учебных курсов, а сервис учебных материалов отвечает за создание и получение самих материалов, а также предоставляет возможность загружать связанные с ними файлы.

### 3.3.4. Уровень представления

Уровень представления в проекте реализован в виде REST API c использованием Python-фреймворка FastAPI[[1]](#footnote-1). Этот слой является точкой входа для всех внешних взаимодействий с системой (запросы с клиента). Его основная задача – принимать HTTP-запросы, преобразовывать их в вызовы соответствующей бизнес-логики на прикладном уровне и возвращать пользователю ответ. Рассмотрим реализацию точки API для создания курса, приведенный в листинге 10.

router = APIRouter(

prefix=settings.api.courses,

tags=["Courses"],

)

@router.post("/", response\_model=CoursePostResponse, status\_code=201)

async def create\_course(

request: CoursePostRequest,

courses\_service: CoursesService = Depends(get\_courses\_service),

):

try:

course = await courses\_service.create\_course(request.name, request.description, request.semester, request.teacher, request.hours)

return CoursePostResponse(

data=course

)

except Exception as e:

print(traceback.format\_exc())

raise HTTPException(

status\_code=400,

detail="Error creating course"

)

Листинг 10 – Точка API для получения курсов

Изначально происходит валидация входных данных, параметр request, которому передается схема CoursePostRequest, указывает FastAPI, что тело HTTP POST-запроса должно соответствовать Pydantic-схеме CoursePostRequest. Также получаем сервисы курсов, для дальнейшего делегирования действий прикладному уровню. Результат, полученный от сервиса, упаковывается в схему ответа CoursePostResponse и отправляется клиенту.

## 3.4. Клиентская часть

### 3.4.1. Атомная архитектура

Основной задачей клиентской части приложения является формирование интуитивно понятного пользовательского интерфейса. Для достижения этой цели был выбран подход, вдохновленный принципами Atomic Design.

На самом базовом уровне архитектуры компонентов находятся “атомы” – фундаментальные строительные блоки, которые не подлежат дальнейшей декомпозиции. К атомам можно отнести компонент FeaturedStar, который в пользовательском интерфейсе является звездой, с помощью которой можно добавить курс в избранные. Данный атом-компонент представлен в листинге 11.

interface Props {

active: boolean

onClick: MouseEventHandler

}

export default const FeaturedStar: FC<Props> = ({ active, onClick }) => {

const Icon = active ? StarFilled : StarOutlined

const color = active ? 'gold' : '#999'

return (

<Icon

style={{

color,

marginLeft: '0.5rem',

fontSize: '16px',

cursor: 'pointer',

}}

onClick={onClick}

/>

)

}

Листинг 11 – Атом компонент для звезды

Следующий уровень иерархии занимают «молекулы». Молекулы представляют собой относительно простые, но уже функционально осмысленные группы атомов, объединенные для выполнения конкретной небольшой задачи или формирования логически связанного блока интерфейса.

Примером молекулы в клиентской часть проекта является элемент меню для курсов (CoursesMenuItem). Задача этого компонента – предоставить элемент, представляющий один конкретный курс в списке навигации. Чтобы выполнить эту задачу, CoursesMenuItem объединяет в себе несколько элементов атомов: текст, который является названием для курса, а также иконка звездочки, которую мы также рассмотрели в предыдущем пункте. Объединяя эти два атома – текстовое название и иконку, а также добавляя к ним общую обертку, получаем CoursesMenuItem. Эта комбинация уже не просто набор отдельных частей, а единый, функциональный элемент навигации, который может быть легко использован для каждого курса, который необходимо отобразить в меню.

Поднимаясь на следующий уровень в иерархии Atomic Design, мы переходим к «организмам». Организмы представляют собой значительно более крупные и сложные компоненты пользовательского интерфейса. Они формируются путем осмысленного объединения различных молекул и, в некоторых случаях, отдельных атомов, которые вместе образуют относительно самостоятельные, функционально завершенные секции или блоки на странице.

Компонент меню для курсов (CourseMenu) является наглядным примером такого организма. Он играет ключевую роль в навигационной системе, предоставляя пользователю доступ к перечню учебных курсов. Во-первых, CourseMenu динамически отображает список всех курсов, для каждого из которых используется компонент-молекула CoursesMenuItem, которая уже была рассмотрена ранее. Для этого CourseMenu получает массив данных о курсах, который извлекается из глобального хранилища состояний.

На основе этого массива данных, CourseMenu итеративно генерирует и отображает экземпляры CoursesMenuItem, передавая каждому из них необходимые атрибуты, такие как название курса, а также его уникальный идентификатор (для формирования навигационной ссылки). Во-вторых, если пользователь аутентифицирован, то у него появляется возможность добавить курс в избранное, что также влияет на логику работы компонента CourseMenu, потому что он извлекает данные о добавленных в избранные курсах из глобального хранилища и динамически отображает также элементы меню и для избранных курсов конкретного пользователя.

На следующем уровне абстракции находятся «шаблоны». Шаблоны определяют общую структуру и раскладку различных типов страниц в приложении, но они еще не содержат конкретного контента или данных. Они служат своего рода «скелетом» для будущих страниц. В проекте эту роль выполняют компоненты, расположенные в директории layout. Центральным элементом этой системы является компонент MainLayout, который выступает в роли главного шаблона для большинства страниц приложения. MainLayout определяет высокоуровневую структуру интерфейса, логически разделяя его на основные зоны: верхнюю часть – шапку (реализованную компонентом Header), боковую навигационную панель (компонент Sider) и центральную область, предназначенную для отображения основного контента текущей страницы Content. В листинге 12 представлен шаблон.

export default const MainLayout: FC<Props> = ({ children }) => {

return (

<Layout>

<Header />

<Layout>{children}</Layout>

</Layout>

)

}

Листинг 12 – Шаблон приложения MainLayout

Вершиной иерархии Atomic Design являются «страницы». Страницы – это конкретные реализации шаблонов, наполненные реальными данными, логикой и функциональностью. Именно страницы видит пользователь при взаимодействии с приложением. Они напрямую связаны с системой маршрутизации и отвечают за отображение информации, соответствующей определенному URL-адресу. Компоненты страниц размещены в директории page.

К таким компонентам в проекте относятся HomePage, LoginPage, RegisterPage, MaterialPage и AuthPage. Каждый из этих файлов представляет собой уникальный компонент-страницу, который, тем не менее, использует общий шаблон MainLayout.

Рассмотрим, например, страницу HomePage, которая является домашней страницей. Она придерживается уже описанного общего шаблона, которому передается соответствующее логике страницы заполнение. Пример HomePage продемонстрирован в листинге 13.

export default const HomePage: FC = () => {

return (

<MainLayout>

<Sider>

<CourseMenu />

</Sider>

<Content />

</MainLayout>

)

}

Листинг 13 – Домашняя страница

### 3.4.2. Обработка API

Взаимодействие клиентской части приложения с бэкенд-системой осуществляется посредством HTTP-запросов, для управления которыми используется библиотека Axios. Логика API-коммуникаций инкапсулирована в специализированных асинхронных функциях, сгруппированных по типу ресурсов (например, auth, courses, materials) в директории api.

Первоначально необходимо было сделать конфигурированный экземпляр Axios. Эта предварительная настройка HTTP-клиента позволяет определить общие параметры, которые будут применяться ко всем исходящим API-запросам. К таким параметрам относятся базовый URL-адрес серверной части (/api/)

Каждый файл в директории отвечает за конкретную точку API. Важным аспектом является строгая типизация запросов и ожидаемых ответов с использованием TypeScript-интерфейсов, определенных в api.ts (например, ApiResponse<T>) и data.ts (например, Course), что повышает надежность кода. В листинге 14 приведен пример обработки точки API для получения списка курсов.

export default async function apiGetCourses(): Promise<Course[]> {

try {

const response = await requester.get<ApiResponse<Course[]>>(

COURSES\_LIST,

{},

)

console.log('response courses', response)

return response.data.data

} catch (error) {

throw error

}

}

Листинг 14 – Получение курсов

### 3.4.3. Обработка состояния приложения

Для управления состоянием приложения используется библиотека Zustand, которая позволяет создать централизованное хранилище, доступное из любого компонента приложения. Конфигурация и определение структуры глобального хранилища находятся в файле globalStore. Интерфейс API, предоставляемый библиотекой Zustand, позволяет определить состояние и функции для его обновления. Это помогает избежать проблемы "протаскивания пропсов" (props drilling). Структура глобального хранилища в приложении включает несколько ключевых сегментов данных:

1. Информацию о текущем аутентифицированном пользователе (или undefined, если пользователь не вошел в систему).
2. Массив, хранящий общий список всех доступных учебных курсов.
3. Массив курсов, отмеченных пользователем как избранные.
4. Объект, представляющий текущий выбранный пользователем курс.
5. Массив учебных материалов (вероятно, загружаемых для выбранного курса).
6. Объект, представляющий текущий выбранный учебный материал.

Для каждого из этих полей состояния определены соответствующие функции-действия (actions), такие как setUser, setCourses, setSelectedCourse и т.д. Эти actions представляют собой функции, которые принимают новые данные и обновляют соответствующую часть хранилища с помощью функции set, предоставляемой Zustand. В листинге 15 частично представлено глобальное хранилище.

interface IGlobalStore {

user: UserData | undefined

setUser: (user: UserData | undefined) => void

courses: Course[]

setCourses: (courses: Course[]) => void

// ... другие поля и actions ...

}

const useGlobalStore = create<IGlobalStore>(set => ({

user: undefined,

setUser: user => set({ user }), // Пример action для обновления данных пользователя

courses: [],

setCourses: courses => set({ courses }), // Пример action для обновления списка курсов

// ... реализация для остальных полей ...

}))

Листинг 15 – Глобальное хранилище

Компоненты приложения получают доступ к данным из хранилища и к функциям-действиям, используя хук useGlobalStore, это позволяет компонентам как отображать актуальную информацию (например, имя пользователя или список курсов), так и инициировать изменения в глобальном состоянии (например, после успешного входа пользователя, загрузки данных с сервера через API или выбора пользователем какого-либо элемента в интерфейсе).

### 3.4.4. Провайдеры

Для предоставления определенной глобальной функциональности, независимо от их уровня вложенности и без необходимости передачи пропсов через промежуточные компоненты, в проекте используется механизм контекста. Логика инкапсулирована в специализированных компонентах-провайдерах, которые находятся в директории providers. Ключевым провайдером в приложении является AuthProvider, роль которого заключается в управлении жизненным циклом пользовательской сессии и инициализации основных данных приложения. AuthProvider выполняет следующие действия:

1. Проверяет статус аутентификации текущего пользователя.
2. По результатам проверки обновляет информацию о пользователе в глобальном хранилище.
3. Загружает начальные данные, необходимые для работы приложения, такие как общий список курсов и список избранных курсов для аутентифицированного пользователя (также сохраняя их в хранилище).

Другим важным провайдером является NotificationProvider, который предоставляет централизованную систему для отображения всплывающих уведомлений. Это позволяет любому компоненту приложения легко и единообразно выводить сообщения для пользователя, информируя его об успешном выполнении операций, возникших ошибках или других значимых событиях.

## 3.5. Хранение файлов

Система хранения файлов обеспечивает загрузку, безопасное сохранение и доступ к файлам учебных материалов. Она состоит из нескольких важных компонентов, каждая из которых выполняет свою важную роль.

Главным параметром является STATIC\_DIR – это путь к директории файловой системе удаленного сервера, где будут располагаться все загруженные файлы. Определение этого пути происходит при запуске backend-приложения. Для этого используется класс StaticSettings, который принимает значения параметра из env файла. (см. листинг 16)

class StaticSettings(BaseSettings):

STATIC\_DIR: str

model\_config = SettingsConfigDict(

env\_file="static.env", env\_file\_encoding="utf-8")

def get\_static\_dir(self) -> str:

return self.STATIC\_DIR

Листинг 16 – Конфигурация файлового хранилища

Когда пользователь инициирует загрузку файла, он взаимодействует с клиентской частью приложения, которое отправляет файл на бэкенд, где запрос обрабатывается соответствующей точкой API. FastAPI автоматически преобразует данные в удобный объект UploadFile, который содержит файл (поток байтов), а также его метаданные.

Далее управление переходит сервисному слою MaterialsService, который проверяет корректность запроса, подготавливает метаинформацию о файле (генерирует уникальный идентификатор, определяет имя файла для сохранения и для отображения пользователю).

Самым основным шагом является вызов метода upload\_file у файлового репозитория, которому передается как Pydantic-модель MaterialFile, содержащая все собранные метаданные, так и сам объект UploadFile.

За запись файла отвечает репозиторий файлов StaticRepository. В первую очередь формируется полный абсолютный путь для сохранения файла, комбинируя сконфигурированный путь STATIC\_DIR и уникальное имя файла (которое пришло из file\_url от сервиса), также проводится проверка безопасности, чтобы предотвратить запись файла за пределы разрешенной директории. Затем содержимое файла сохраняется в файловом хранилище и, что очень важно, StaticRepository модифицирует поле file\_url в переданном ему объекте MaterialFile, добавляя к уникальному имени файла префикс /static/. Таким образом file\_url из, например, xxxx.pdf превращается в /static/xxxx.pdf и обновленный объект MaterialFile возвращается обратно сервису. (см. листинг 17)

async def upload\_file(self, file\_data: UploadFile, file: MaterialFile) -> Optional[MaterialFile]:

try:

file\_path = os.path.abspath(

os.path.join(self.static\_dir, file.file\_url))

if not file\_path.startswith(os.path.abspath(self.static\_dir)):

raise ValueError("Invalid file path")

os.makedirs(os.path.dirname(file\_path), exist\_ok=True)

with open(file\_path, "wb") as buffer:

shutil.copyfileobj(file\_data.file, buffer)

file.file\_url = f"/static/{file.file\_url}"

return file

except Exception as e:

print(f"Error uploading file: {e}")

return None

Листинг 17 – Загрузка файла в хранилище

После того, как файл успешно сохранен и его URL сформирован, метаинформация вместе с этим URL сохраняется в базе данных с помощью репозитория MaterialRepository, который работает с ORM-моделью MaterialFileModel.

Когда пользователь хочет получить (скачать) файл, процесс выглядит иначе. Фронтенд-приложение, получив от бэкенда информацию о материалах, также получает информацию о прикрепленных файлах, включая их file\_url и file\_name. На основе этих данных фронтенд генерирует для пользователя ссылку, при клике на которую браузер отправляет HTTP GET-запрос на указанный путь. Этот запрос перехватывает и обрабатывается механизмом StaticFiles фреймворка FastAPI. Для этого в главном файле main настроена специальная “точка монтирования”, которая приведена в листинге 18.

app.mount("/static", StaticFiles(directory=settings.static.get\_static\_dir()), name="static")

Листинг 18 – Загрузка файла в хранилище

Проще говоря, данная строка сообщает FastAPI, что все запросы, у который путь начинается с /static, должны обрабатываться как запросы к статическим файлам. Искать эти файлы нужно в директории, которая прописана в env файле (о ней шла речь выше). StaticFiles находит запрошенный файл в упомянутой выше директории и отдает содержимое клиенту.

## 3.6. Искусственный интеллект

Основной задачей данного модуля является генерация информативных ответов и использованием нейросетевой модели gpt-4o-mini, которая предоставляется API OpenAI. Качество отчетов, а также их релевантность повышается за счет обогащения входных данных для LLM информацией, извлекаемой из внутренней базы данных.

Важным этапом является извлечение и обработка контекста из базы данных, то есть производится анализ вопроса на наличие предопределенных ключевых слов. Данный анализ помогает определить потребности пользователя, а именно запрашивает ли он информацию о преподавателе, учебных часах или конкретных материалов. Далее программа пытается идентифицировать курс, по которому пользователь задает вопрос, путем проверки вхождения имени курса в вопрос пользователя. Если курс найден, то формируется начальная часть контекста. Далее уже в зависимости от того, что именно запрашивает пользователь, в контекст добавляется дополнительная информация, полученная из базы данных. Если на этапе определения курса, он не был найден, но запрос включал в себя ключевые слова, то в контекст добавится строка о том, что информации по данному курсу нет или нужно сформулировать запрос более подробно.

После получения строки контекста, программа определяет, как именно взаимодействовать с API OpenAI. Если данная строка соответствует одному из общих или неинформативных сообщений, то в таком случае в OpenAI передается только исходный запрос пользователя. В ином случае, если контекст содержательный, то формируется запрос и пользовательское сообщение, объединяющие извлеченный контекст и вопрос.

# 4. Тестирование и анализ результатов

Для тестирования REST API использовался инструмент Swagger, который предоставляет интерактивную документацию API, которая позволяет разработчикам взаимодействовать с API непосредственно через веб-интерфейс. Swagger UI генерирует веб-страницу, на которой представлены все доступные точки API, сгруппированные по тегам. Для каждой точки указаны: HTTP-метод (GET, POST, PUT, DELETE), путь (URL), ожидаемые параметры (тело запроса), возможные ответы с сервера.

На рисунке 5 изображены все точки API, использованные в приложении.

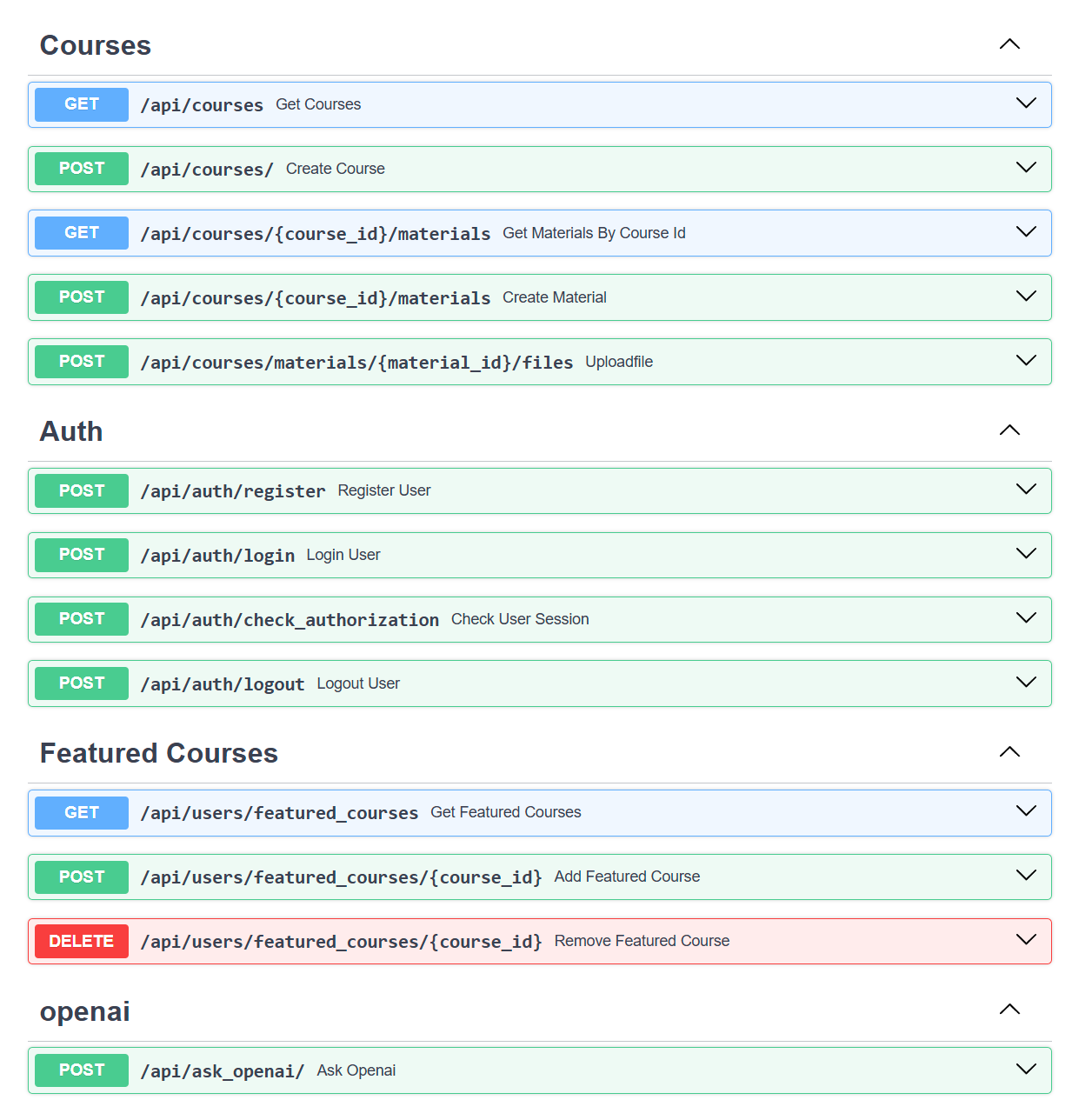


Рисунок 5 – Точки API

Рассмотрим результаты тестирования для некоторых из них. Листинг 19 демонстрирует результат запроса на получение всех имеющихся курсов в базе данных.

{

"data": [

{

"id": 301,

"name": "Операционные системы",

"description": "Фундаментальный курс, охватывающий...",

"semester": 1,

"teacher": "Брагин Алексей Владимирович",

"hours": 144

},

...

]

}

Листинг 19 – Ответ API на запрос получения всех курсов

Листинг 20 демонстрирует результат запроса на получение материалов конкретного курса “Операционные системы”.

{

"data": [

{

"id": 38,

"material\_type": "lectures",

"name": "Поиск минимума функции одного ...

"number": 1,

"content": "Лекция посвящена...",

"files": [

{

"id": "5536fbf9-2564-4c18-8376-e767a0797dbe",

"material\_id": 38,

"file\_name": "lecture\_1.pdf",

"file\_description": "string",

"file\_url": "/static/5536fbf9-2564-4c18-8376-e767a0797dbe.pdf"

},

...

]

},

...

]

}

Листинг 20 – ответ API на запрос получения всех курсов

Листинг 21 демонстрирует результат запроса на регистрацию пользователя. При успешной регистрации API возвращает созданного пользователя, а также данные будут сохранены в базу данных (данные из базы данных отображены в таблице № 13.

{

"data": {

"username": "bmstu",

"email": "bmstu@mail.ru"

}

}

Листинг 21 – ответ API на регистрацию пользователя

Таблица 13 – данные о зарегистрированном пользователе в базе данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | email | hashed\_password | username | is\_teacher |
| 86 | bmstu@mail.ru | $2b$12$... amnfi | bmstu | true |

Рассмотрим результаты работы программы. На рисунке 6 изображена домашняя страница, на которой представлено боковое навигационное меню, содержащее все курсы, а также контент, содержащий основную информацию по выбранному курсу, и шапка сайта.

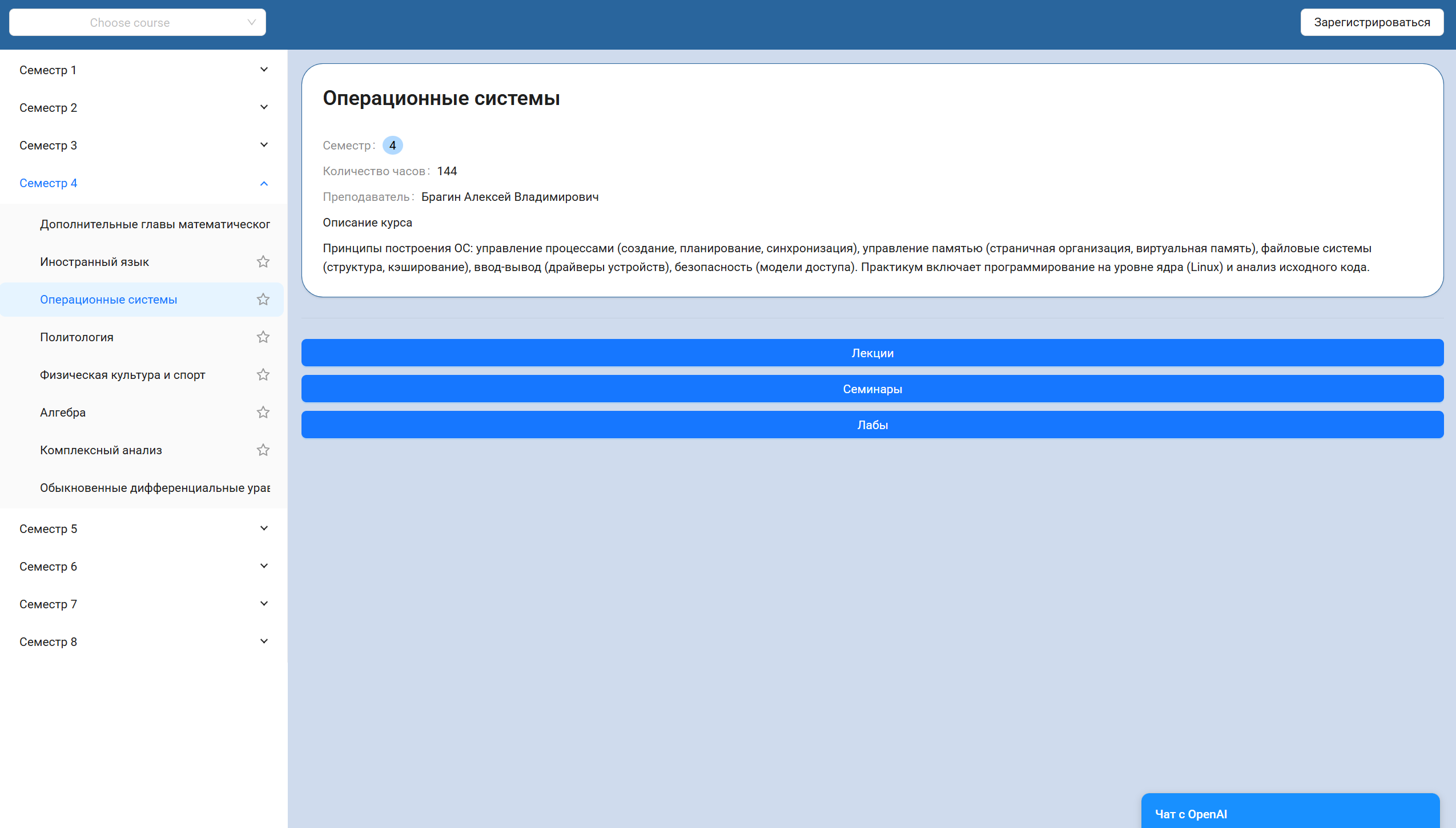


Рисунок 6 - Домашняя страница

Чтобы получить доступ к дополнительным возможностям сайта – необходимо зарегистрироваться или, при наличии аккаунта, войти в систему. На рисунке 7 и 8 изображена регистрация и логин пользователя.

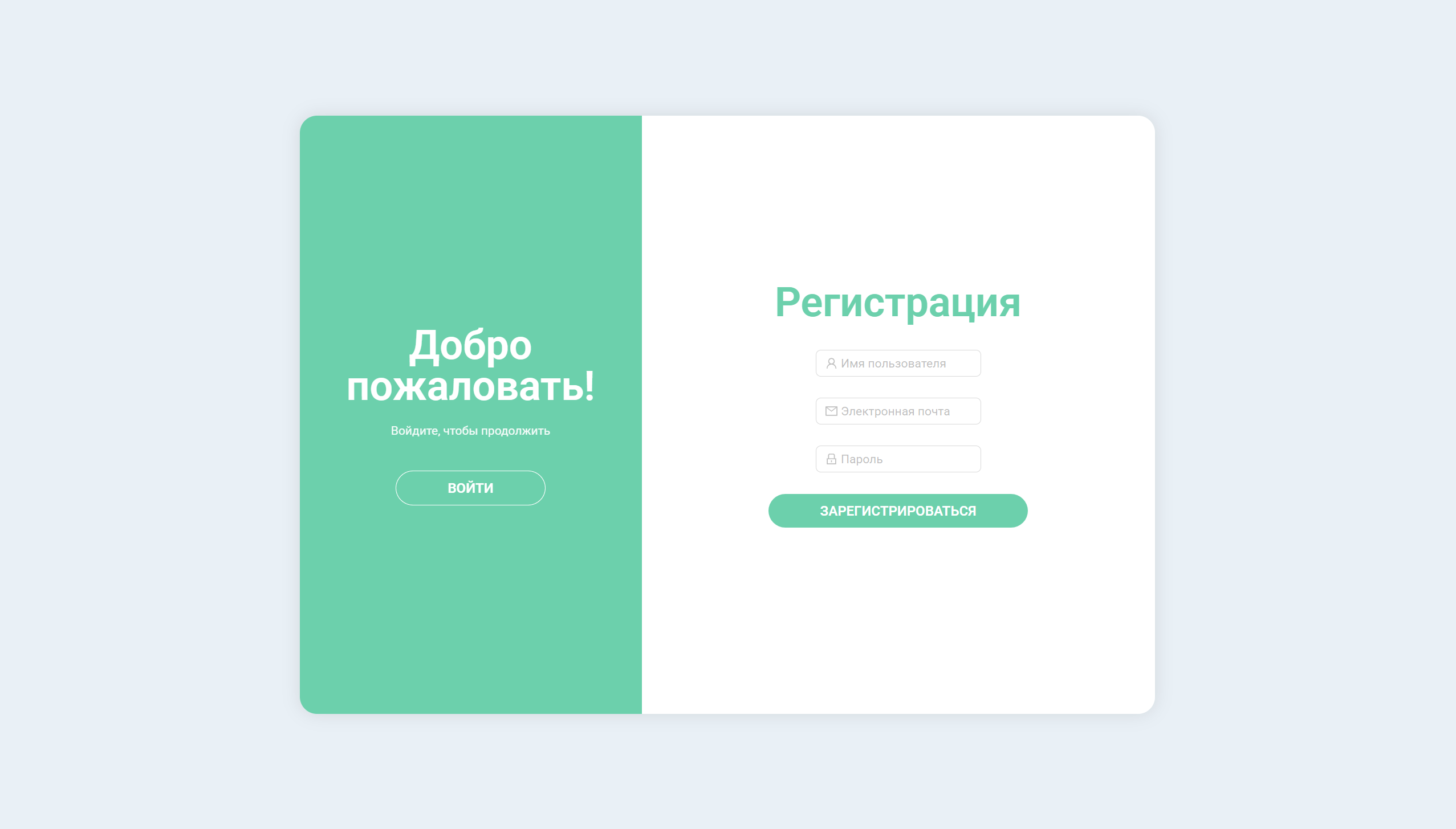


Рисунок 7 – Страница регистрации

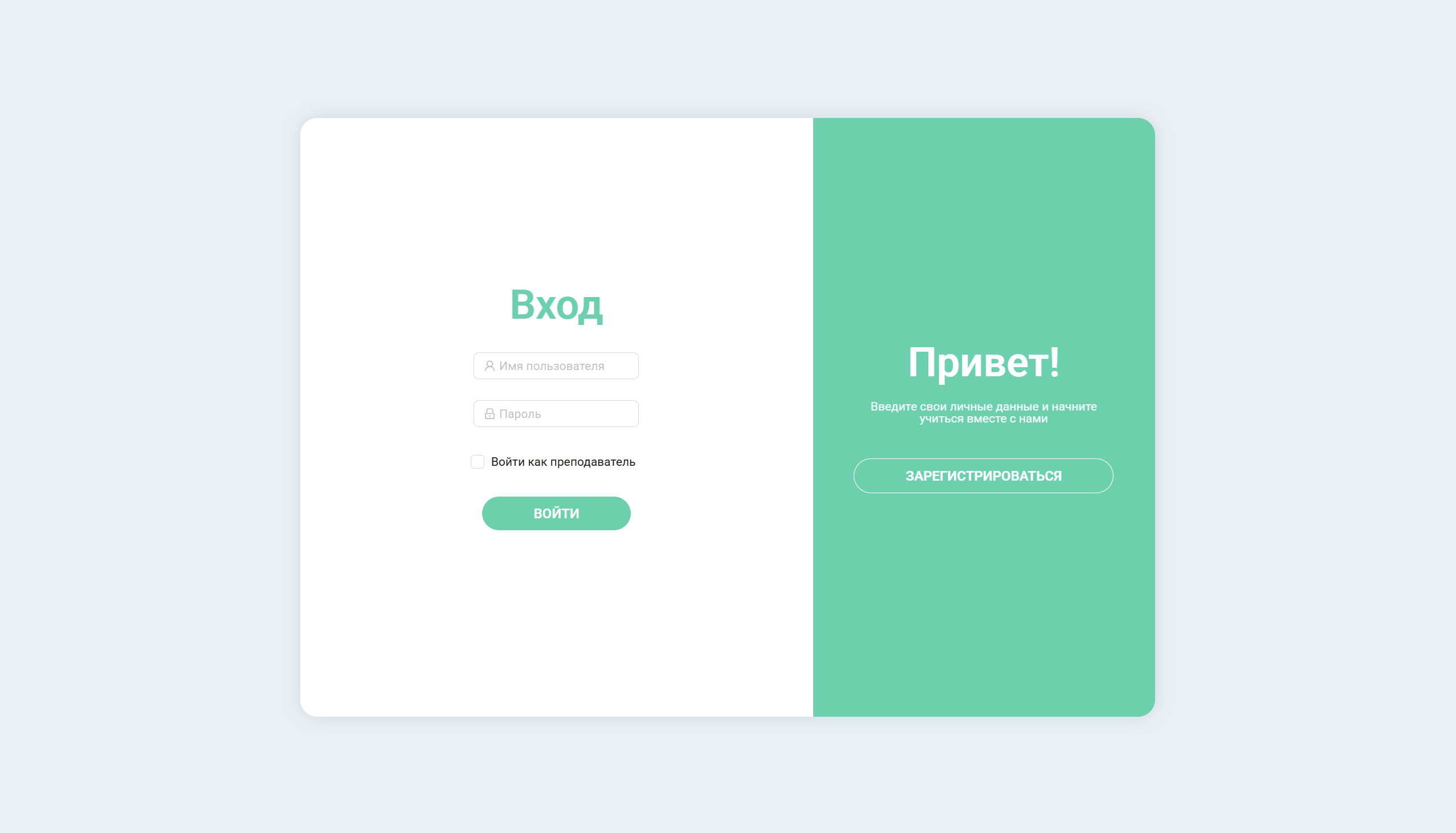


Рисунок 8 – Страница входа в систему

Далее функциональность сайта разделяется на две части в зависимости от того, кто им пользуется – студент или преподаватель. Если студент пользуется данным сервисом, то после регистрации ему предоставлена возможность пользоваться материалами, ИИ-ассистентом, а также студент сможет добавлять курсы в избранное, для более удобного пользования. (см рисунок 9-10).

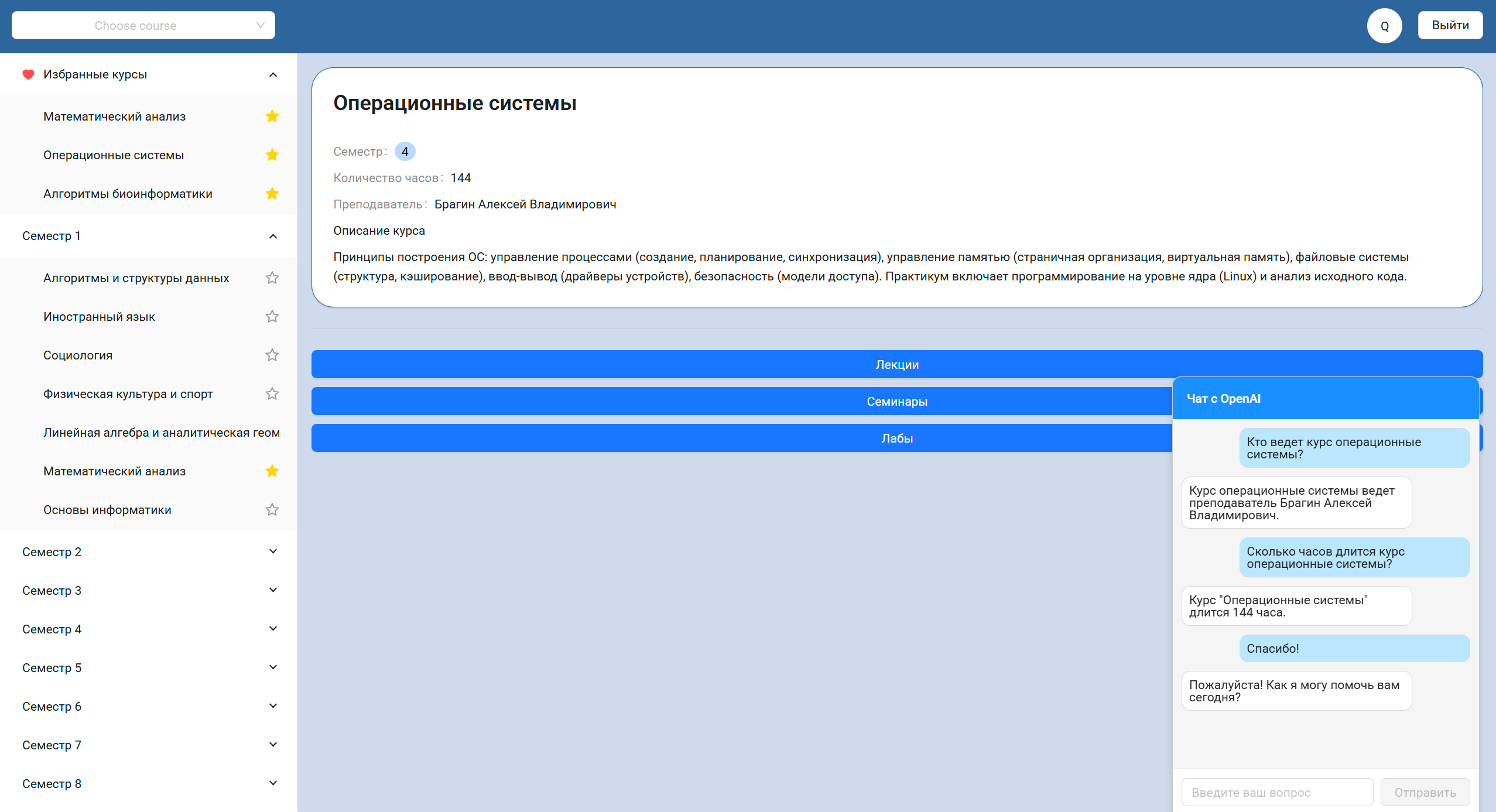


Рисунок 9 – Домашняя страница для студента

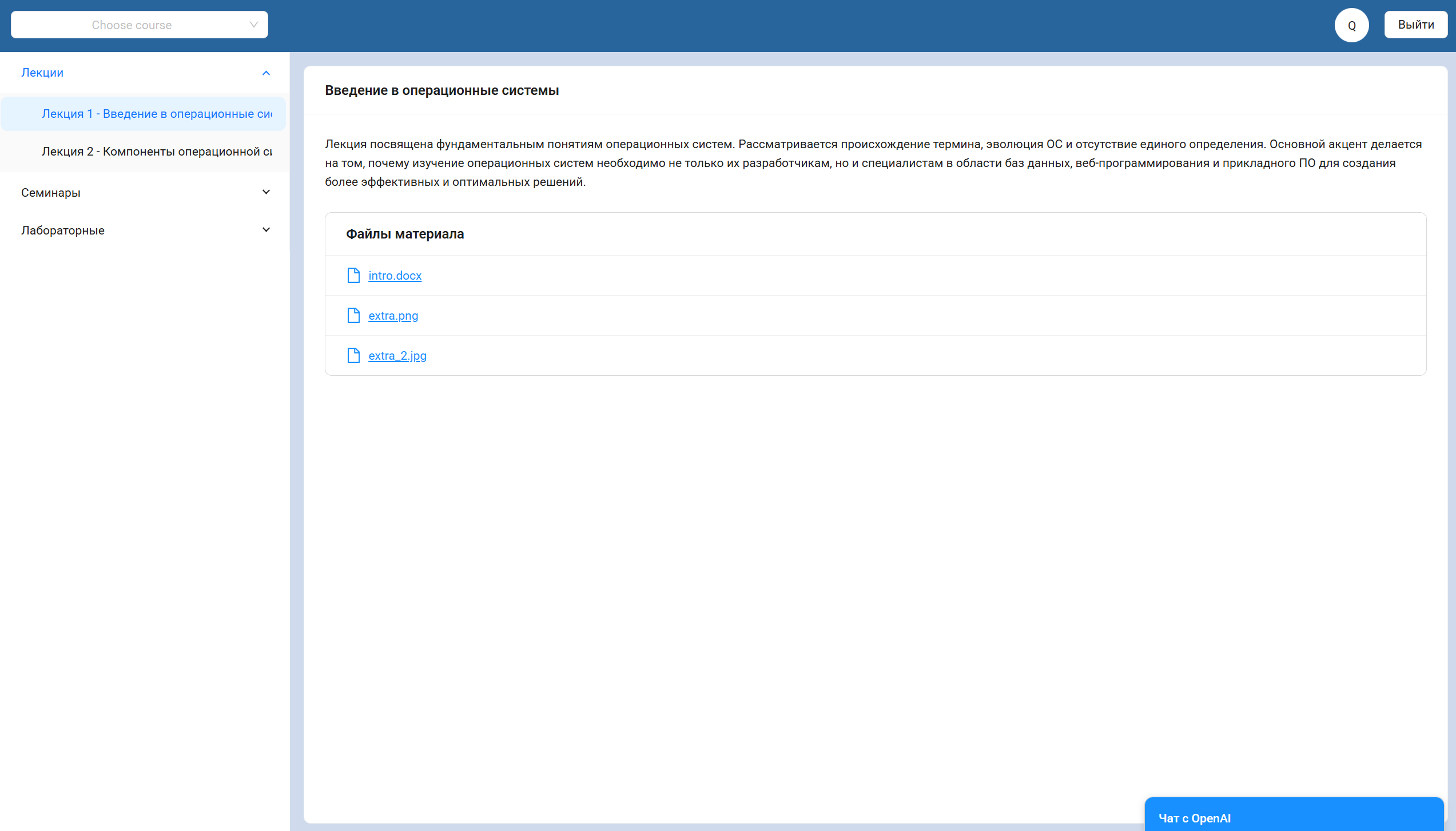


Рисунок 10 – Страница с материалами

Для преподавателей созданы специальные учетные записи, с помощью которых они смогут пользоваться также дополнительными сервисами, чтобы добавлять новые курсы, материалы и файлы. Чтобы добавить новые курсы, необходимо ввести название курса, его описание, семестр и количество часов, которое занимает курс. (см. рисунок 11). Для добавления новых материалов, в выпадающем списке нужно выбрать курс, для которого добавляем материалы, а также ввести название материал, его тип (лекция, семинар, лабораторная или экзамен), номер и содержание. (см. рисунок 12). Аналогично, для уже существующего материала можно добавить любого формата файлы. Необходимо выбрать только из выпадающего списка курс, материал и ввести название и описание файла, а также прикрепить его. (см. рисунок 13).

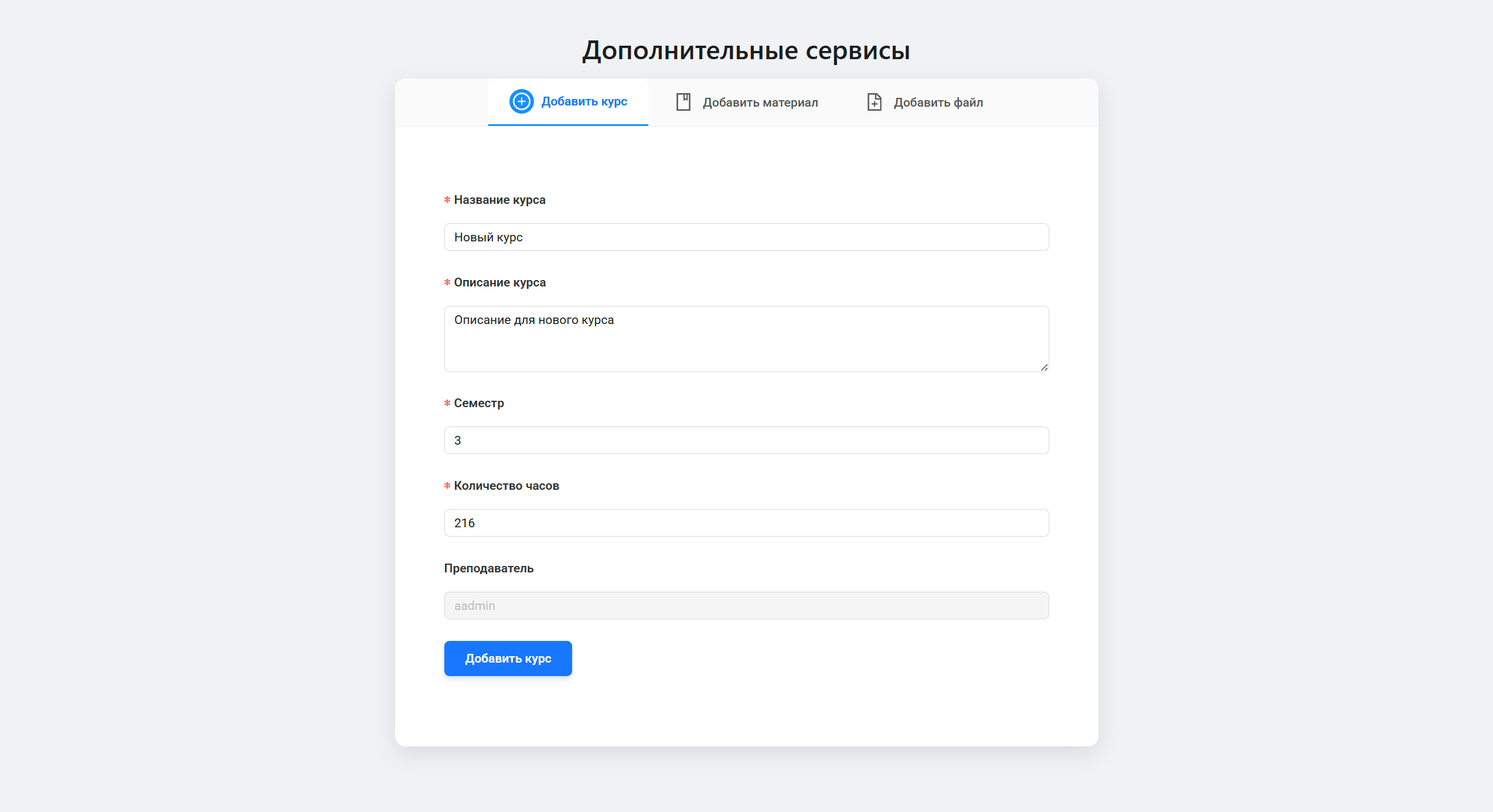


Рисунок 11 – Страница для добавления новых курсов

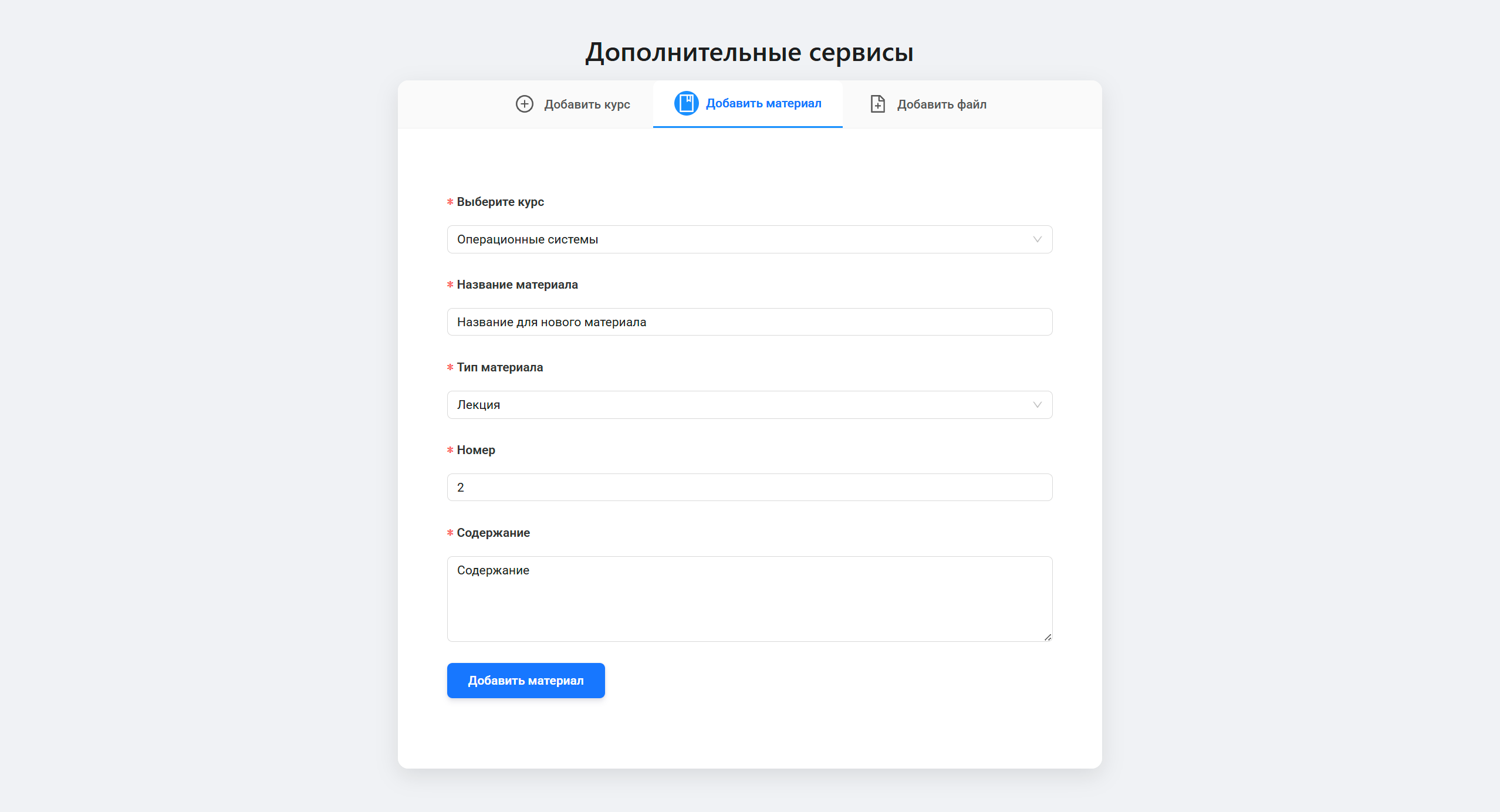


Рисунок 13 – Страница для добавления новых материалов

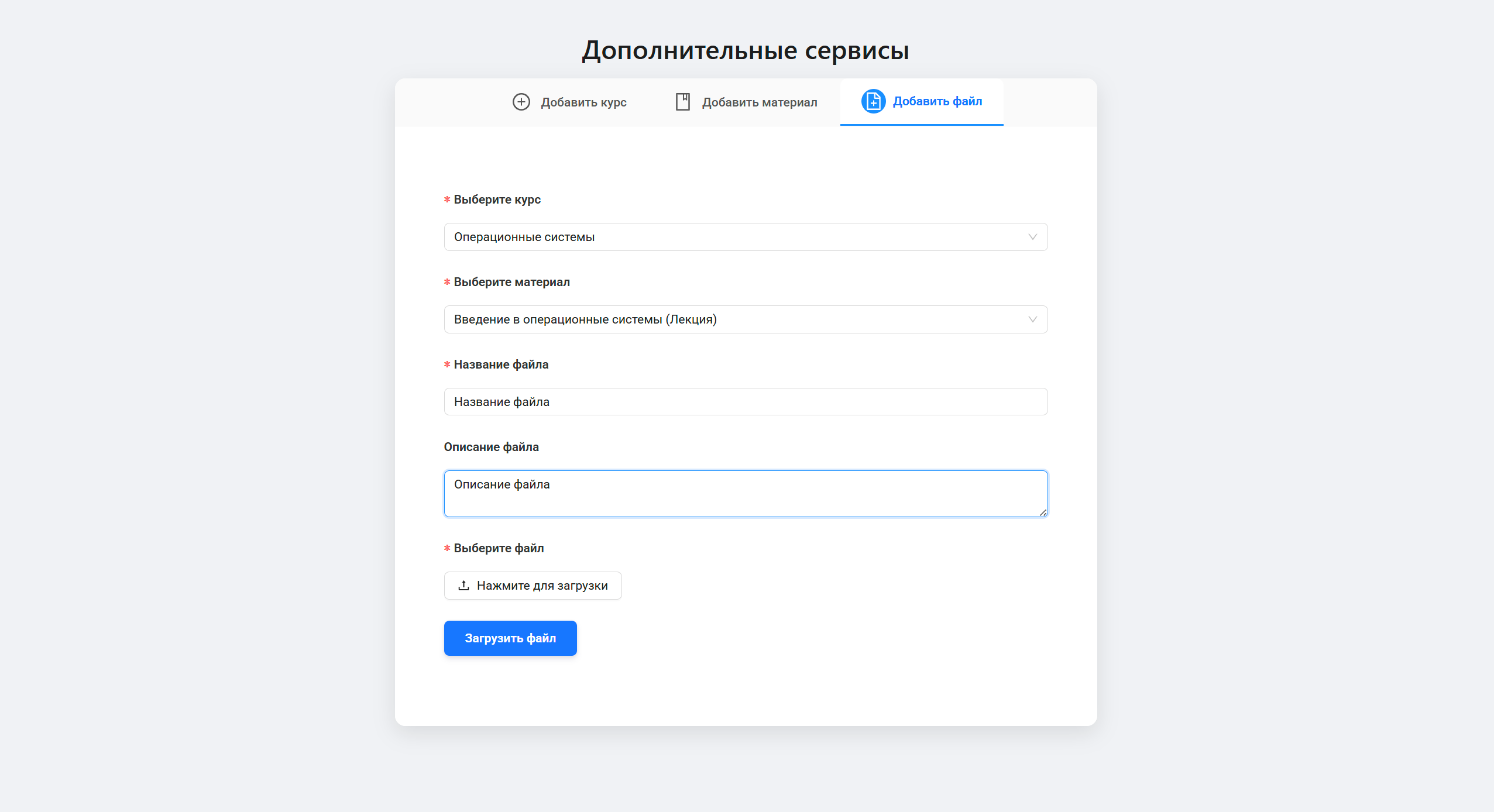


Рисунок 14 – Страница для добавления новых файлов

Также во всем приложении поддерживается система нотификации. При удачном входе в систему, добавлении курса в избранное или возникновении ошибки будет всплывать уведомление, сообщающее пользователю эту информацию. Примеры таких уведомлений представлены на рисунках 15-16.

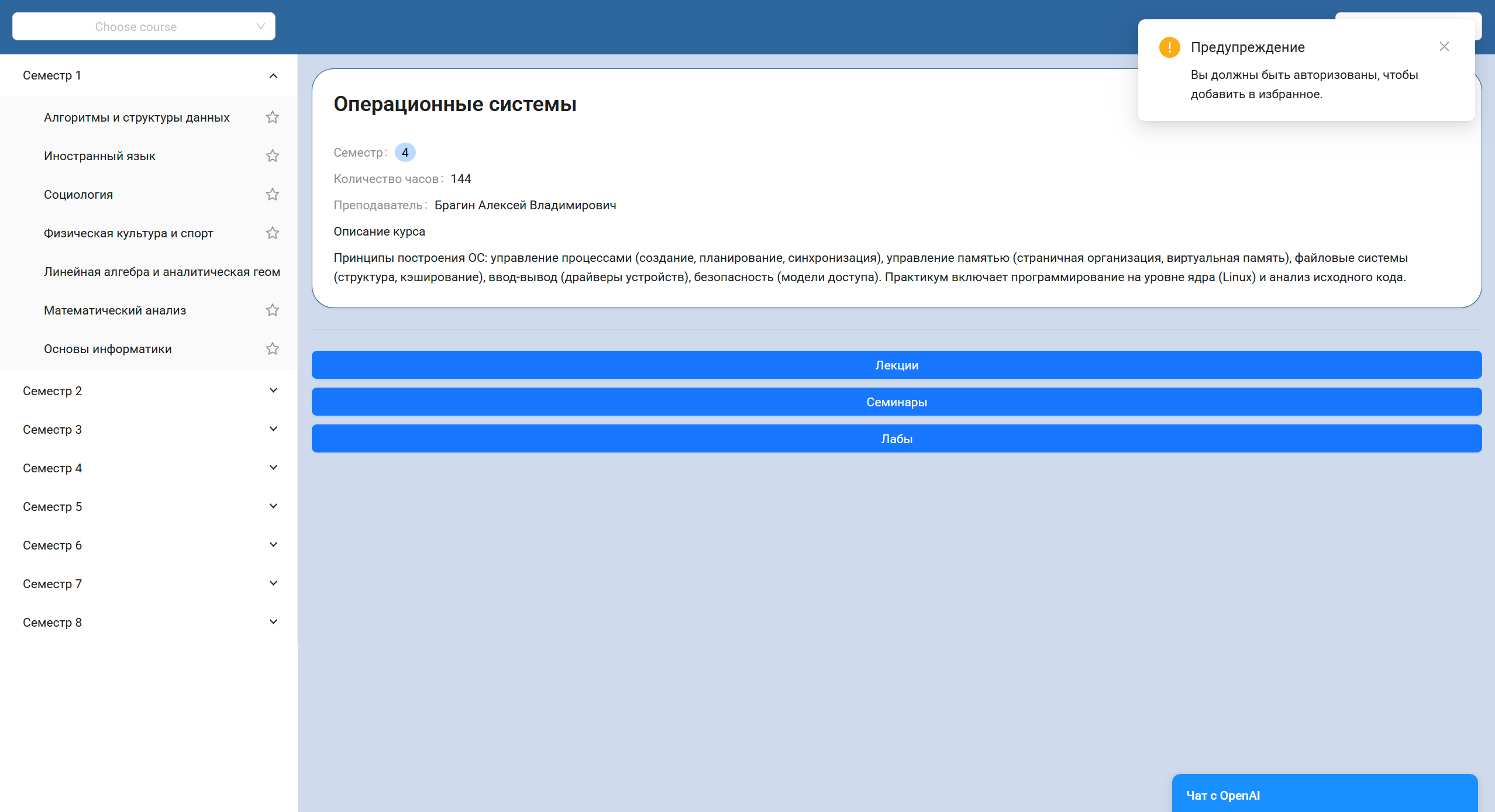


Рисунок 15 – Уведомление о необходимости авторизации

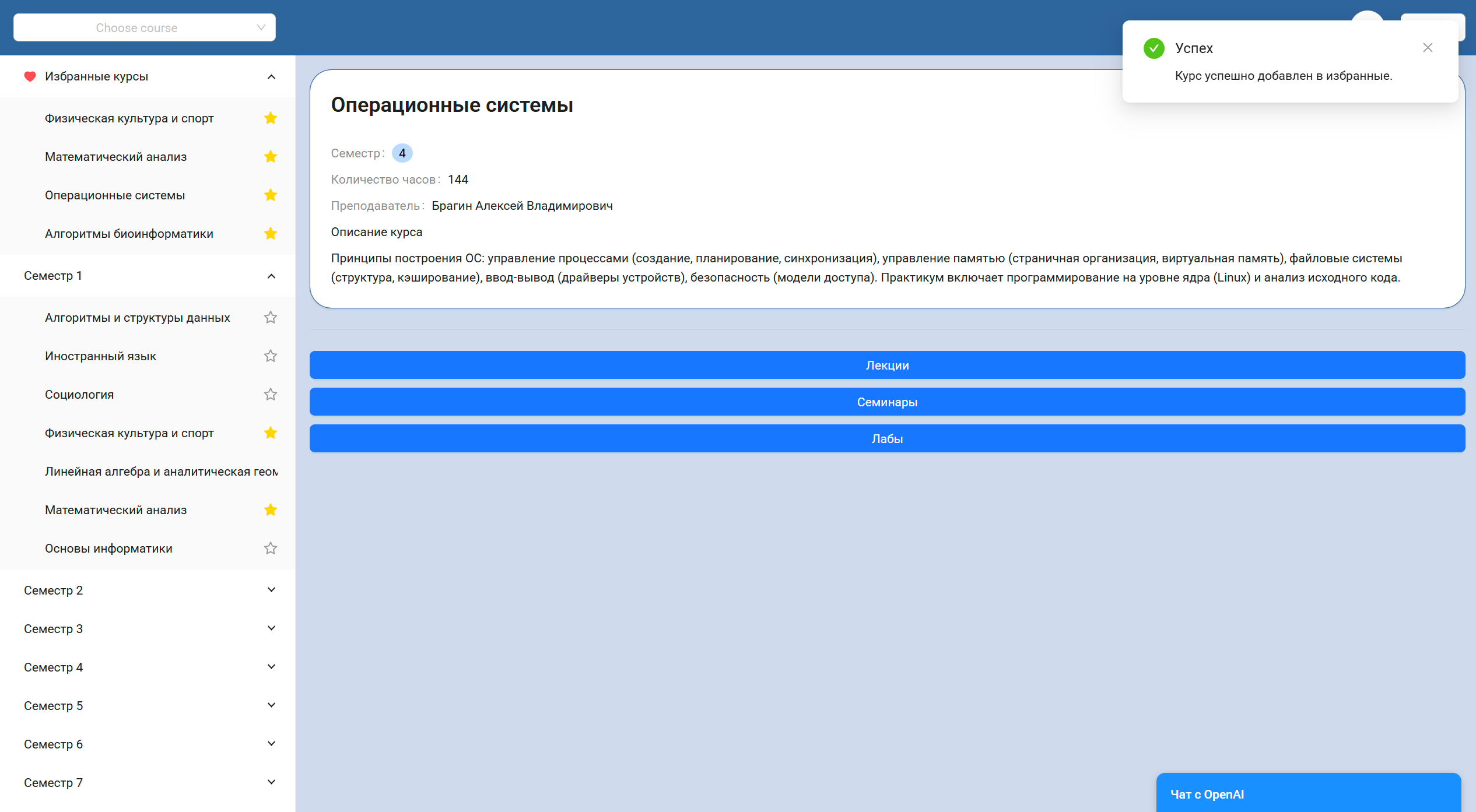


Рисунок 16 – Уведомление об успешном добавлении курса в избранное

# ЗАКЛЮЧЕНИЕ

Основной целью проекта являлось создание централизованного источника учебной информации для студентов кафедры ИУ9 МГТУ им. Н.Э. Баумана, которая была успешно достигнута. В ходе выполнения данной работы были решены все поставленные задачи по разработке веб-приложения системы управления базой знаний для студентов, включая интеграцию с технологиями искусственного интеллекта. Разработанное приложение представляет собой комплексную систему, состоящую из клиентской и серверной частей. Клиентская часть обеспечивает интуитивно понятный и удобный пользовательский интерфейс. Серверная часть реализует многоуровневую архитектуру, включающую инфраструктурный, доменный, прикладной и представительный уровни, что обеспечивает надежность, масштабируемость и потенциал дальнейшего развития и доработки системы.

Ключевым элементом системы стала интеграция с нейросетевой моделью gpt-4o-mini через API OpenAI. Разработанный механизм обогащения запросов контекстом, извлеченным из внутренней базы данных учебных материалов, позволяет модели генерировать ответы, основанные не на общедоступной информации из интернета, а на проверенных и утвержденных материалах кафедры. Система также включает функционал для загрузки, хранения и управления файлами учебных материалов. В результате проделанной работы создано полнофункциональное веб-приложение, которое успешно справляется с задачами предоставления доступа к учебным курсам, материалам, а также предлагает интеллектуального ассистента для поддержки учебного процесса.

В результате проведённого тестирования были проверены все функции системы, отдельно проведено тестирование серверной части по запросам, а также отдельно протестирована клиентская часть на различных браузерах и устройствах. Тестирование подтвердило корректное решение всех поставленных задач.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Brown, T. B. Language Models are Few-Shot Learners / T. B. Brown, B. Mann, N. Ryder [et al.] // arXiv. – 2020. – arXiv:2005.14165. – URL: https://arxiv.org/abs/2005.14165 (дата обращения: 24.03.2025). – Текст: электронный.
2. Lewis, P. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks / P. Lewis, E. Perez, A. Piktus [et al.] // arXiv. – 2020. – arXiv:2005.11401. – URL: https://arxiv.org/abs/2005.11401 (дата обращения: 15.05.2025). – Текст: электронный.
3. Мартин, Р. Ч. Чистая архитектура. Искусство разработки программного обеспечения / Р. Ч. Мартин; пер. с англ. А. Слинкина. – Санкт-Петербург: Питер, 2018. – 352 с. – (Библиотека программиста). – ISBN 978-5-496-0 чистая-архитектура-978-5-496-02 чистая-архитектура-978-5-496-0.
4. Эванс, Э. Предметно-ориентированное проектирование (DDD). Построение сложных программных систем / Э. Эванс; пер. с англ. С. Пасербы. – Москва: Вильямс, 2011. – 448 с. – (Addison-Wesley Signature Series). – ISBN 978-5-8459-1593-2.
5. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер; пер. с англ. под ред. Д. Михеева. – Москва: Вильямс, 2015. – 544 с. – (Addison-Wesley Signature Series). – ISBN 978-5-8459-2028-4.
6. React: The library for web and native user interfaces / Meta and community contributors. – Текст: электронный // React: [сайт]. – 2024. – URL: https://react.dev/ (дата обращения: 11.03.2025).
7. Фрост, Б. Атомарный дизайн / Б. Фрост. – Текст: электронный // Брэд Фрост: [сайт]. – 2016. – URL: https://bradfrost.com/blog/post/atomic-design/ (дата обращения: 12.04.2025).
8. Nginx Documentation. Reverse Proxy Guide / Nginx. – Текст: электронный // Nginx: [сайт]. – 2025. – URL: https://nginx.org/en/docs/ (дата обращения: 03.05.2025).
9. TypeScript Handbook / Microsoft. – Текст: электронный // TypeScript: [сайт]. – 2024. – URL: https://www.typescriptlang.org/docs/ (дата обращения: 06.03.2025).
10. Ant Design: The world's second most popular React UI library / Ant Group. – Текст: электронный // Ant Design: [сайт]. – 2025. – URL: https://ant.design/ (дата обращения: 12.05.2025).
11. Python 3.12 documentation / Python Software Foundation. – Текст: электронный // Python.org: [сайт]. – 2025. – URL: https://docs.python.org/3/ (дата обращения: 10.04.2025).
12. Bayer, M. SQLAlchemy: The Python SQL Toolkit and Object Relational Mapper / M. Bayer, SQLAlchemy contributors. – Текст: электронный // SQLAlchemy Project: [сайт]. – 2025. – URL: https://www.sqlalchemy.org/ (дата обращения: 14.05.2025).

1. Исходный код библиотеки по адресу https://github.com/fastapi/ [↑](#footnote-ref-1)