ОБЛАСТНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

**«ТОМСКИЙ ЭКОНОМИКО-ПРОМЫШЛЕННЫЙ КОЛЛЕДЖ»**

К ЗАЩИТЕ ДОПУЩЕН

Директор ОГБПОУ «ТЭПК»

\_\_\_\_\_\_\_\_\_\_\_\_ / О.Н. Пояркова

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

**ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ "Библиотека ИИ и его эффективное использование"**

ДИПЛОМНЫЙ ПРОЕКТ

*Специальность 09.02.07 Информационные системы и программирование*

Выполнил:

Студент IV курса группы 1912с

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Волков Д.П.

Проверил:

Руководитель КДП

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Уляхин В.А.

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Томск 2025

Оглавление

[ГЛАВА 1. ТЕХНИЧЕСКОЕ ЗАДАНИЕ 3](#_Toc200831444)

[Актуальность 3](#_Toc200831445)

[Описание предметной области 3](#_Toc200831446)

[Общее описание проекта 4](#_Toc200831447)

[Ролевая модель 4](#_Toc200831448)

[Диаграмма прецедентов (Диаграмма вариантов использования) 5](#_Toc200831449)

[Диаграмма деятельности (Диаграмма активностей) 7](#_Toc200831450)

[Диаграмма последовательности 10](#_Toc200831451)

[Технологический стек 13](#_Toc200831452)

[ГЛАВА 2. Проектирование и разработка базы данных 14](#_Toc200831453)

[Концептуальная модель данных 14](#_Toc200831454)

[Логическая модель данных 15](#_Toc200831455)

[Физическая модель данных 16](#_Toc200831456)

[Словарь данных 18](#_Toc200831457)

[ГЛАВА 3. МАКЕТ-ДИЗАЙН веб-приложения 22](#_Toc200831458)

[UI Kit 22](#_Toc200831459)

[Повторяющиеся блоки интерфейса 23](#_Toc200831460)

[Главная страница 24](#_Toc200831461)

[Страница статьи 25](#_Toc200831462)

[ГЛАВА 4. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ 25](#_Toc200831463)

[Начальная настройка серверной части 25](#_Toc200831464)

[Генерация основных компонентов серверной логики 27](#_Toc200831465)

[Маршрутизация 28](#_Toc200831466)

[Миграции и наполнение базы данных 29](#_Toc200831467)

[Контроллеры 30](#_Toc200831468)

[Модели 41](#_Toc200831469)

[ГЛАВА 5. РАЗРАБОТКА Веб части 47](#_Toc200831470)

[ЗАКЛЮЧЕНИЕ 48](#_Toc200831471)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 49](#_Toc200831472)

# **ГЛАВА 1. ТЕХНИЧЕСКОЕ ЗАДАНИЕ**

## Актуальность

## Описание предметной области

Предметная область охватывает деятельность виртуальной библиотеки, предоставляющей доступ к материалам, посвящённым искусственному интеллекту (в дальнейшем «ИИ») и его эффективному применению. Основная цель библиотеки — предоставить список ИИ и предоставить пользователям возможность обмениваться знаниями, а именно как практично и эффективно пользоваться ИИ в различных сферах деятельности.

Библиотека объединяет ИИ с различным назначением, обсуждения и статьи, написанные пользователями. Важной функцией веб-приложения является организация информации, позволяющая пользователям быстро находить нужные материалы и обмениваться опытом в рамках сообщества и свободно размещать новый материал.

**Цель**

Разработать и спроектировать веб-приложение "Библиотека ИИ и его эффективное использование", обеспечивающее удобный доступ к материалам, а также предоставляющее платформу для взаимодействия пользователей между собой.

**Задачи**

1. Анализ предметной области;
2. Проектирование архитектуры веб-приложения:
   * Создать информационную структуру, обеспечивающую интуитивную навигацию;
   * Разработать дизайн пользовательского интерфейса (UI).
3. Реализация функционала веб приложения:
   * Разработать разделы веб-приложения, включая каталог ИИ, личный кабинет пользователя, статьи, обсуждения.
4. Тестирование и доработка веб-приложения:
   * Провести автоматизированное тестирование со стороны сервера на соответствие требованиям, выявить и устранить возможные ошибки;
   * Провести ручное тестирование веб-сайта на соответствие требованиям, выявить и устранить возможные ошибки.

## Общее описание проекта

Все пользователи системы подразделяются на три группы:

1. Администраторы;
2. Пользователи;
3. Гости.

Гости могут авторизироваться и регистрироваться. Авторизация производится по токенам, генерируемые при авторизации и отзываемых при выходе. Авторизация на другом устройстве не должна вызывать «вылет» из аккаунта на всех устройствах.

Администратор форума имеет возможность удалить комментарии пользователя, но **не изменять.** Свободно взаимодействовать со статьями форума, ИИ и обсуждениями (т.е. изменять, удалять информацию), а также создавать новую информацию.

Пользователи получают доступ к некоторым функциям только после успешной авторизации.

Функционал администратора:

1. CRUD операции статей;
2. CRUD операции обсуждений;
3. CRUD операции записей библиотеки ИИ;
4. CRUD операции комментариев (Изменение только своих комментариев).

Функционал пользователя:

1. Редактирование профиля;
2. Просмотр статей;
3. Просмотр обсуждений;
4. Просмотр библиотеки ИИ;
5. Создание и редактирование созданной им статьи;
6. Создание и редактирование созданного им обсуждения;
7. Создание и редактирование созданной им записи ИИ;
8. Создание и редактирование созданного им комментария.

Функционал гостя:

1. Авторизация;
2. Регистрация;
3. Просмотр статей;
4. Просмотр обсуждений;
5. Просмотр библиотеки ИИ.

## Ролевая модель

Пользователь — основной авторизированный пользователь платформы, который создаёт, размещает информацию об ИИ.

Гость — неавторизированный пользователь, которому доступна только регистрация и вход в аккаунт.

Администратор —авторизированный пользователь платформы, который редактирует информацию об ИИ и следит за сообществом на платформе.

В таблице 1 показано распределение прав доступа для каждой роли:

Таблица 1. Ролевая модель

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Раздел** | **Фичи** | **Админ** | **Пользователь** | **Гость** |
| Навигация | Раздел статьи | да | да | да |
| Раздел обсуждения | да | да | да |
| Раздел библиотека ИИ | да | да | да |
| Профиль | да | да | нет |
| Авторизация | Вход | нет | нет | да |
| Регистрация | нет | нет | да |
| Выход | да | да | нет |
| Библиотека ИИ | Просмотр | да | да | да |
| Создание | да | да | нет |
| Удаление | да | нет | нет |
| Редактирование | да | да | нет |
| Статьи | Просмотр | да | да | да |
| Удаление | да | нет | нет |
| Редактирование | да | да | нет |
| Создание | да | да | нет |
| Создание комментария | да | да | нет |
| Редактирование комментария | да | да | нет |
| Удаление комментария | да | да | нет |
| Обсуждения | Просмотр | да | да | да |
| Удаление | нет | нет | нет |
| Редактирование | да | да | нет |
| Создание | да | да | нет |
| Создание комментария | да | да | нет |
| Редактирование комментария | да | да | нет |
| Удаление комментария | да | да | нет |
| Профиль | Редактирование | да | да | нет |
| Просмотр профиля | да | да | нет |

## Диаграмма прецедентов (Диаграмма вариантов использования)

Диаграмма прецедентов — это визуальное представление взаимодействия пользователей (акторов) с системой. Она показывает основные функции системы (прецеденты) и то, как акторы их используют. Диаграмма помогает понять, какие действия доступны для каждой роли и как они связаны между собой (см. рисунок 1).

Ключевые элементы и их взаимодействие:

**Акторы (роли):**

* Гость: Неавторизованный пользователь;
* Пользователь: Авторизованный пользователь;
* Администратор: Управляющий.

**Прецеденты (функции):**

* Для гостя: авторизация, регистрация, просмотр записей библиотеки, статей и обсуждений;
* Для пользователя: просмотр профиля, редактирование профиля, создание обсуждения, изменение статуса обсуждения, создание комментария, изменение комментария, удаление комментария, создание записи в библиотеке, создание статей, редактирование записи в библиотеке, редактирование статьи;
* Для администратора: просмотр профиля, редактирование профиля, создание обсуждения, изменение статуса обсуждения, создание комментария, изменение комментария, удаление комментария, создание записи в библиотеке, создание статей, удаление статьи, изменение статьи, изменение записи в библиотеке, удаление записи в библиотеке.

**Взаимодействие:**

* Акторы взаимодействуют с системой через прецеденты;
* Используются связи include (обязательное включение одного прецедента в другой) и extend (дополнительное расширение функционала).

Гость может просматривать статьи, библиотеку и обсуждения, но для создания обсуждения или комментария ему нужно стать пользователем через авторизацию или регистрацию.

Администратор может выполнять все функции пользователя, но имеет больше прав на управление контентом (статьями, обсуждениями, записями библиотеки, комментариями).

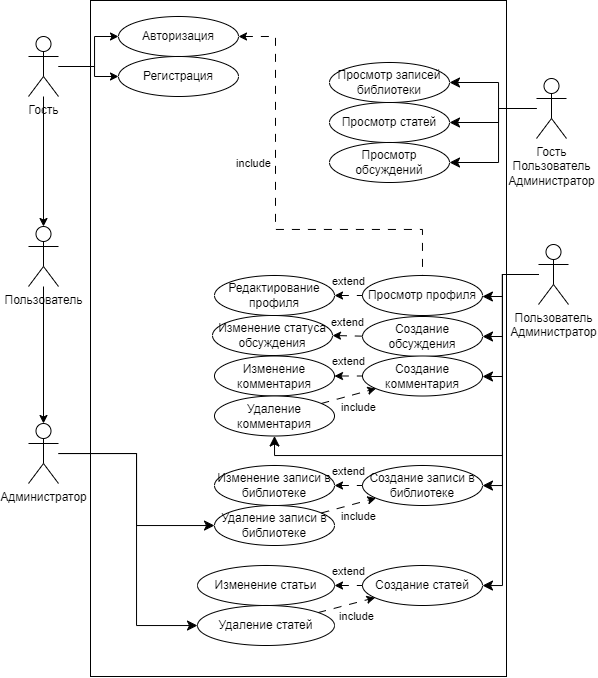


Рисунок 1 - Диаграмма прецедентов

## Диаграмма деятельности (Диаграмма активностей)

Диаграмма деятельности (Диаграмма активностей) — это визуальное представление процессов или алгоритмов в системе, которое показывает последовательность действий и поток управления между ними. Она используется для моделирования рабочих процессов, бизнес-процессов или логики выполнения операций в системе.

Диаграммы показывают основные процессы взаимодействия пользователя и системы на сайте. Каждая диаграмма отражает этапы выполнения действий пользователя и отклик системы. Рассмотрим ключевые моменты (см. рисунок 2):

1. **Регистрация**:

* **Пользователь** заполняет необходимые поля в форме регистрации;
* **Система** проверяет валидность введённых данных.
  + Если данные невалидны, выводится сообщение об ошибке;
  + Если данные валидны, система выдаёт токен и перенаправляет на главную страницу.

1. **Авторизация:**

* Схожа с регистрацией, но используется форма авторизации;
* При успешной проверке данных пользователь получает токен и переходит на главную страницу;
* При ошибке выводится уведомление.

1. **Выход:**

* Пользователь нажимает кнопку выхода.
* **Система**:
  + Очищает токен;
  + Перенаправляет на главную страницу;
  + Показывает уведомление о завершении операции.

1. **Создание:**

* Пользователь заполняет необходимые данные для создания объекта (например, книги, записи);
* Если данные валидны, система уведомляет о создании;
* В случае ошибки отображается соответствующее сообщение.

1. **Удаление:**

* Пользователь нажимает на кнопку удаления;
* Система удаляет объект и сообщает об успешном завершении.

1. **Редактирование:**

* Пользователь вводит новые данные для редактирования объекта;
* Если данные валидны, происходит успешное обновление, и система уведомляет об этом;
* В случае ошибки пользователь видит сообщение об ошибке.

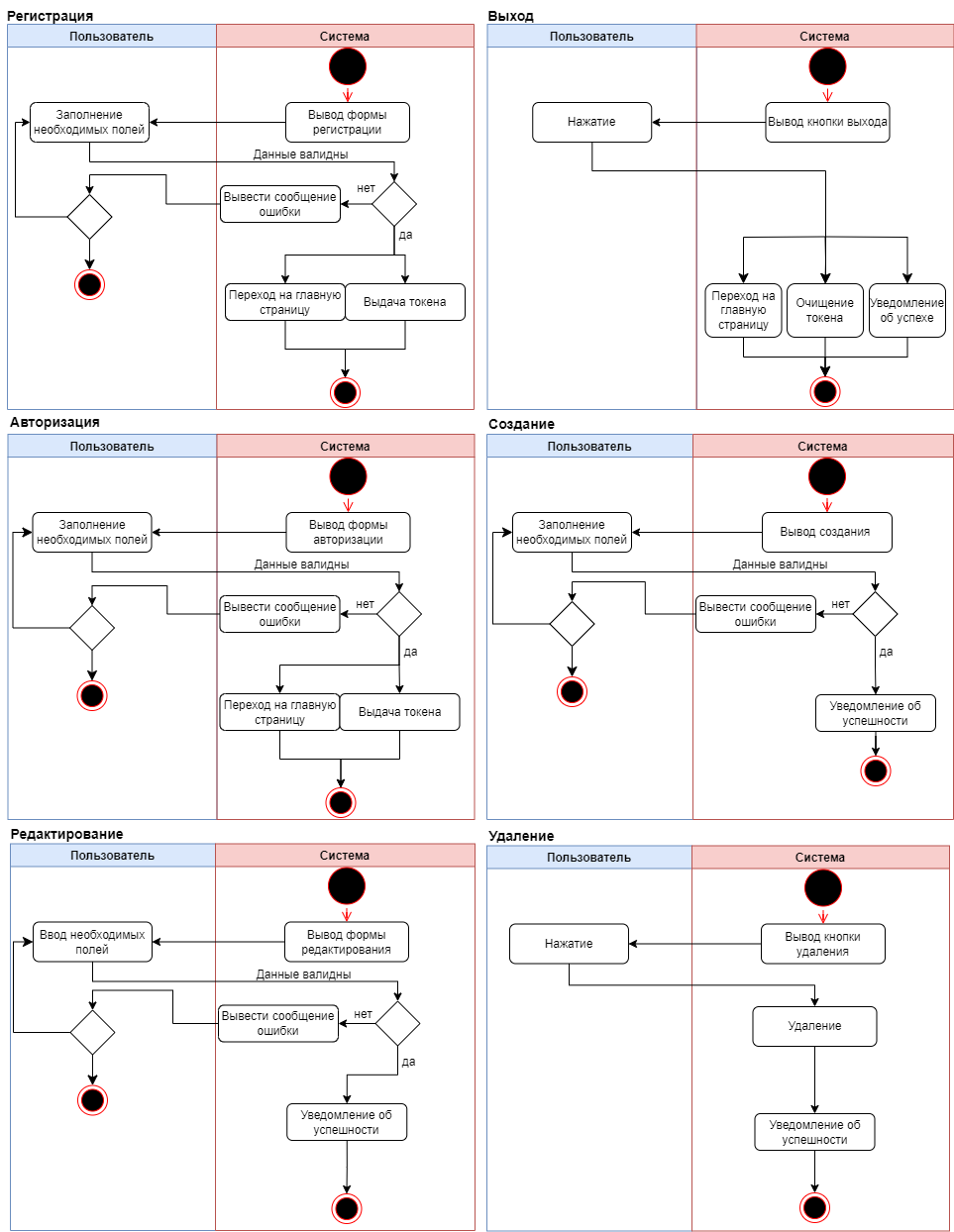


Рисунок 2 - Диаграмма деятельности

## Диаграмма последовательности

Диаграмма последовательности — это вид UML-диаграммы, который показывает взаимодействие объектов в системе в рамках конкретного сценария. Она визуализирует порядок вызовов методов и обмен сообщениями между объектами с течением времени.

Диаграммы представляют процессы взаимодействия пользователя, системы и базы данных на сайте "Библиотека ИИ". Каждая диаграмма иллюстрирует последовательность действий и обмен сообщениями между участниками (пользователь, система, компонент, база данных) (см. рисунок 3).

* + - 1. **Регистрация (Диаграмма 1):**
* Пользователь отправляет запрос на регистрацию.
* **Система**:
  + Отображает форму для ввода данных;
  + Проверяет валидность данных через компонент авторизации;
  + Если данные корректны:
    - Генерируется токен, данные пользователя сохраняются в базе данных;
    - Возвращается успешный ответ (201).
  + Если данные некорректны:
    - Выдаётся сообщение об ошибке (422).

1. **Авторизация (Диаграмма 2):**

* Пользователь отправляет запрос на вход с логином и паролем.
* **Система**:
  + Передаёт данные на проверку в компонент авторизации;
  + Проверка успешна:
    - Пользователь аутентифицирован, токен сохраняется, отправляется успешный ответ (200).
  + Если ошибка в данных:
    - Возвращаются коды ошибок (401 — неверный пароль, 422 — некорректный формат данных).

1. **Выход (Диаграмма 3):**

* Пользователь инициирует запрос на выход.
* **Система**:
  + Удаляет токен пользователя из базы данных;
  + Перенаправляет пользователя на главную страницу;
  + Отправляется успешный ответ (200).

1. **Создание объекта (Диаграмма 4):**

* Пользователь отправляет запрос на создание нового объекта (например, книги).
* **Система**:
  + Проверяет валидность данных;
  + Если данные корректны:
    - Объект создаётся, данные сохраняются в базе;
    - Отправляется успешный ответ (201).
  + Если данные некорректны:
    - Отправляется сообщение об ошибке (422).

1. **Редактирование объекта (Диаграмма 5):**

* Пользователь отправляет запрос на редактирование существующего объекта.
* **Система**:
  + Проверяет доступ к объекту;
  + Если пользователь имеет доступ и данные валидны:
    - Изменения сохраняются в базе;
    - Отправляется успешный ответ (200).
  + Если доступ запрещён или данные некорректны:
    - Выдаётся ошибка (403 или 422).

1. **Удаление объекта (Диаграмма 6):**

* Пользователь инициирует запрос на удаление объекта.
* **Система**:
  + Проверяет права доступа;
  + Если доступ разрешён:
    - Объект удаляется из базы данных;
    - Отправляется успешный ответ (204).
  + Если доступ запрещён:
    - Выдаётся ошибка (403).

Ключевые моменты:

1. **Роли участников:**
   * Пользователь инициирует действия.
   * Система обрабатывает запросы и выполняет операции с базой данных через компонент авторизации.
2. **Обработка данных:**
   * Валидация данных — ключевой этап перед сохранением или изменением информации;
   * Ошибки возвращаются с указанием кода (401, 403, 422 и др.).
3. **Права доступа:**
   * Для операций редактирования и удаления система проверяет права пользователя на объект.
4. **Коды ответов:**
   * Успешные операции возвращают коды 200, 201, 204;
   * Ошибки возвращают соответствующие коды (422, 403, 401).
5. **Повторное использование модулей:**
   * Базовые действия (создание, редактирование, удаление) построены универсально, что позволяет их многократное применение для различных объектов сайта.

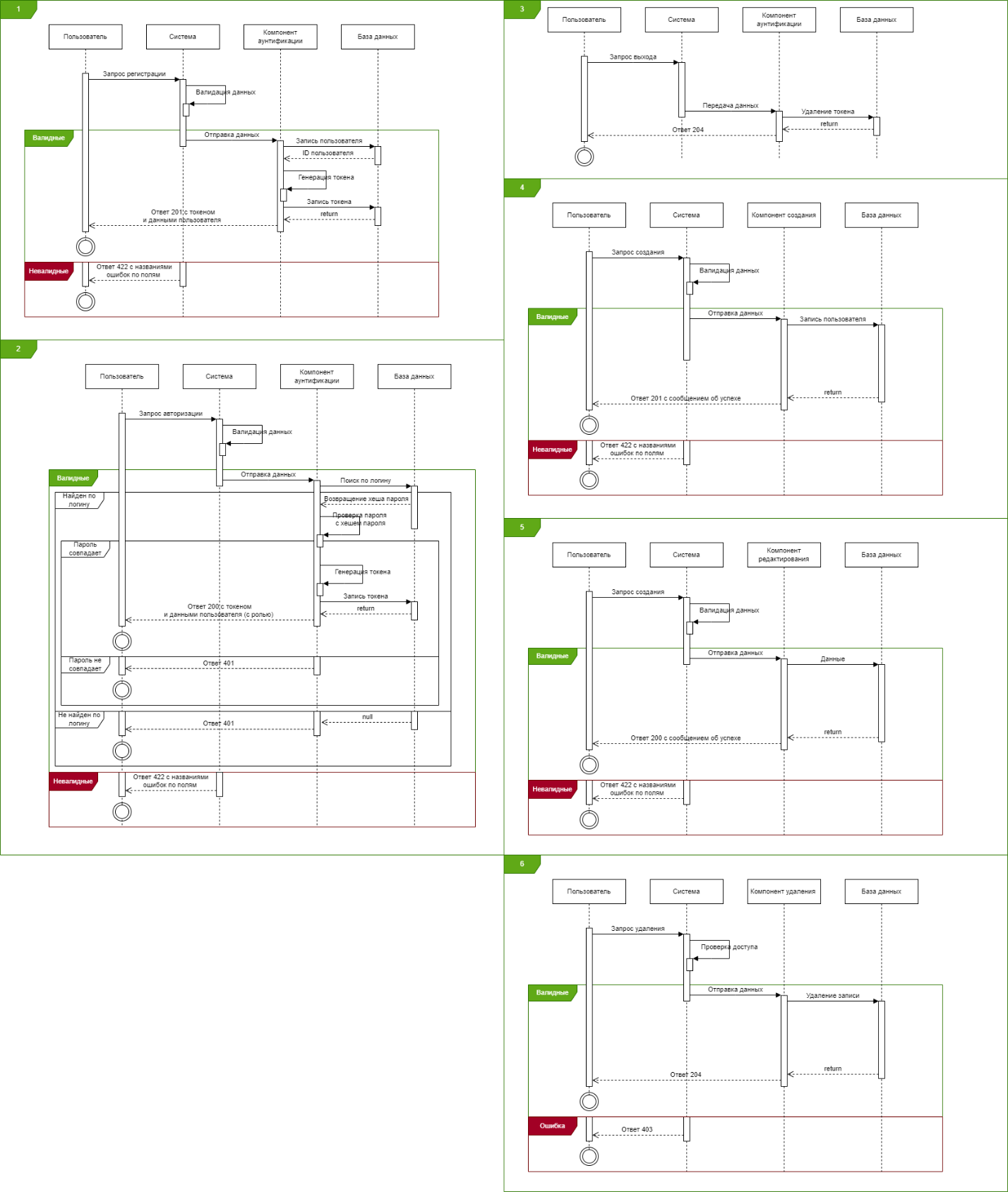


Рисунок 3 - Диаграмма последовательности

## Технологический стек

**Серверная часть**

Язык программирования: PHP 8.2

Фреймворк для разработки: Laravel 11

Система управление базами данными: PhpMyAdmin

Панель управления сервером: Open Server Panel

**Веб-платформа**

Язык программирования и разметки: JavaScrtipt, HTML, CSS

**Инструменты разработки и интеграции**

IDE: PhpStorm

Тестирование: Postman

Создание диаграмм и схем: Draw.io

**Аутентификация и безопасность**

Система аутентификации: Laravel Sanctum

# **ГЛАВА 2. Проектирование и разработка базы данных**

## Концептуальная модель данных

Концептуальная модель — это формализованное описание предметной области, не связанное с какими-либо компьютерными средствами. Сущности (таблицы) обозначаются прямоугольниками; Атрибуты (поля таблиц) — овалы, которые могут быть объедены в скруглённые прямоугольники; Связи с обозначением мощности — стрелками (в направлении, где мощность равна многим — двойная стрелка, а со стороны, где она равна единице — одинарная).

Анализируя предметную область, можно выделить следующие сущности с их атрибутами (см. рисунок 1):

1. Сущность «Роль» с атрибутами «Название», «Код»;
2. Сущность «Пользователь» с атрибутами «Токен», «Почта», «Аватар», «Логин», «Пароль»;
3. Сущность «Статья» с атрибутами «Название», «Текст», «Фото», «Описание»;
4. Сущность «Категория» (Статья) с атрибутами «Название», «Код»;
5. Сущность «Задача» с атрибутами «Название», «Код»;
6. Сущность «Задача\_ИИ»;
7. Сущность «Преобразование» с атрибутами «Название», «Код»;
8. Сущность «Преобразование\_ИИ»;
9. Сущность «ИИ» с атрибутами «Название», «Ссылка», «Платный», «Триал», «Описание»;
10. Сущность «Обсуждение» с атрибутами «Название», «Фото», «Описание», «Статус»;
11. Сущность «Комментарий» с атрибутами «Текст»;
12. Сущность «Категория» (Обсуждение) с атрибутами «Название», «Код».

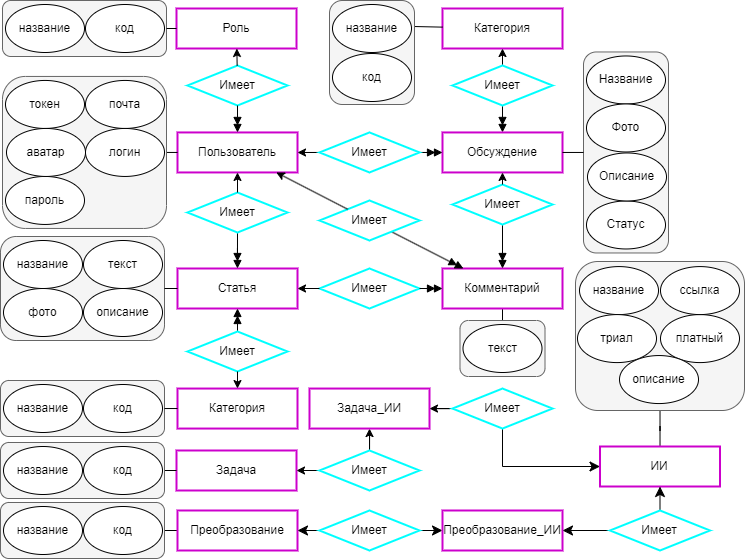


Рисунок 0 - Концептуальная модель данных

## Логическая модель данных

Логическая модель базы данных — схема базы данных, выраженная в понятиях модели данных. Этим отличается от концептуальной модели, описывающей семантику предметной области без указания технологии (конкретных методов реализации), и от физической модели, которая описывает конкретные физические механизмы, применяемые для хранения данных в накопителях.

Используя нашу концептуальную модель данных и выполнив нормализацию данных до 3 нормальной формы включительно, построим логическую модель данных (см.рисунок 2):



Рисунок 0 - Логическая модель данных

## Физическая модель данных

Физическая модель данных — это модель данных, описанная с помощью средств конкретной системы управления базами данных (СУБД). Физическая модель данных строится на базе логической путем добавления особенностей конкретной СУБД. К таким особенностям могут относиться поддерживаемые СУБД типы данных, соглашения о присвоении имен таблицам, атрибутам и т.д. Физическая модель данных фактически является готовым заданием на создание БД, имея которое можно реализовать БД в выбранной СУБД.

Выбранной СУБД является MySQL. Используя логическую модель, построим физическую модель данных (см. рисунок 3):

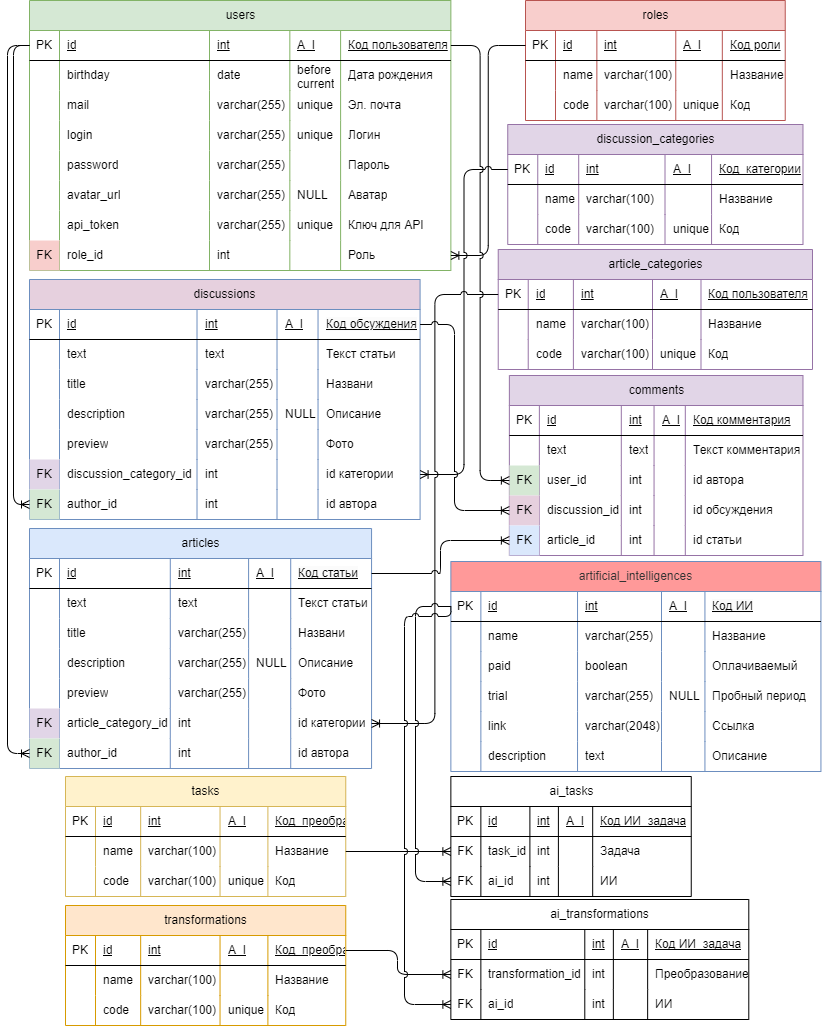


Рисунок 0 - Физическая модель данных

## Словарь данных

Таблица 2 – Словарь данных для БД «Библиотека ИИ и его эффективное использование»

|  |  |  |  |
| --- | --- | --- | --- |
| **users** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор пользователя |
| birthday | date |  | Дата рождения пользователя |
| email | varchar(255) | UNIQUE | Электронная почта пользователя |
| login | varchar(255) | UNIQUE | Логин пользователя |
| password | varchar(255) |  | Пароль пользователя |
| avatar\_url | varchar(255) | NULL | Аватар пользователя |
| role\_id | varchar(255) |  | Роль пользователя |
| **roles** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор роли |
| name | varchar(100) |  | Название роли |
| code | varchar(100) | UNIQUE | Уникальный код роли |
| **ai\_tasks** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор связи AI и задачи |
| task\_id | int | FK (tasks.id) | Идентификатор задачи |
| ai\_id | int | FK (artificial\_intelligences.id) | Идентификатор искусственного интеллекта |
| **ai\_transformations** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор связи AI и преобразования |
| transformation\_id | int | FK (transformations.id) | Идентификатор преобразования |
| ai\_id | int | FK (artificial\_intelligences.id) | Идентификатор искусственного интеллекта |
| **articles** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор статьи |
| text | text |  | Текст статьи |
| title | varchar(255) |  | Заголовок статьи |
| description | varchar(255) | NULL | Описание статьи |
| preview | varchar(255) | NULL | Превью статьи |
| article\_category\_id | int | FK (article\_categories.id) | Категория статьи |
| author\_id | int | FK (users.id) | Автор статьи |
| **article\_categories** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор категории статей |
| name | varchar(100) |  | Название категории |
| code | varchar(100) | UNIQUE | Уникальный код категории |
| **artificial\_intelligences** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор искусственного интеллекта |
| name | varchar(255) |  | Название AI |
| paid | tinyint(1) |  | Является ли платным искусственный интеллект |
| trial | int | NULL | Длительность пробного периода |
| description | text | NULL | Описание искусственного интеллекта |
| link | varchar(2048) | FK (article\_categories.id) | Ссылка на искусственный интеллект |
| **comments** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор комментария |
| text | text |  | Текст комментария |
| user\_id | int | FK (users.id) | Пользователь, оставивший комментарий |
| discussion\_id | int | FK (discussions.id), NULL | Обсуждение, к которому относится |
| article\_id | int | FK (articles.id), NULL | Статья, к которой относится |
| **discussions** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор обсуждения |
| text | text |  | Текст обсуждения |
| title | varchar(255) |  | Заголовок обсуждения |
| description | varchar(255) | NULL | Описание обсуждения |
| preview | varchar(255) | NULL | Превью обсуждения |
| status | tinyint(1) |  | Статус обсуждения |
| discussion\_category\_id | int | FK (discussion\_categories.id) | Категория обсуждения |
| author\_id | int | FK (users.id) | Автор обсуждения |
| **discussion\_categories** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор категории обсуждений |
| name | varchar(100) |  | Название категории |
| code | varchar(100) | UNIQUE | Уникальный код категории |
| **tasks** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор задачи |
| name | varchar(100) |  | Название задачи |
| code | varchar(100) | UNIQUE | Уникальный код задачи |
| **transformations** | | | |
| **Атрибут** | **Тип** | **Ограничение** | **Описание** |
| id | int | PK, AUTO\_INCREMENT | Идентификатор преобразования |
| name | varchar(100) |  | Название преобразования |
| code | varchar(100) | UNIQUE | Уникальный код преобразования |

# **ГЛАВА 3. МАКЕТ-ДИЗАЙН веб-приложения**

Макет-дизайн веб-приложения «Библиотека ИИ и его эффективное использование» был разработан для визуализации ключевых экранов и функциональных блоков системы. Каждое изображение представляет собой часть пользовательского интерфейса, отражающую основные этапы взаимодействия клиента с приложением. Ниже приведено описание каждого макета в соответствии с задачами проекта.

## UI Kit

UI Kit — это визуальное руководство по стилю интерфейса приложения «Библиотека ИИ и его эффективное использование». Он включает в себя базовые элементы дизайна: шрифты, цветовую палитру, кнопки, иконки, карточки и навигационные компоненты (см. Рисунок 4). Эти элементы обеспечивают единообразие оформления всех экранов и ускоряют процесс разработки и тестирования пользовательского интерфейса.

В проекте используется строго структурированная цветовая система:

* Фоновые цвета: от чёрного #000000 до светло-серого #F5F5F5 с промежуточными оттенками #212121, #333333, #444444 — они задают общий тон интерфейса и поддерживают визуальную иерархию блоков;
* Цвета текста: сбалансированные серые #777777, #888888, #CCCCCC и чисто белый #FFFFFF обеспечивают удобочитаемость на тёмных и светлых фонах;
* Акцентные цвета: зелёный #28A745 и красный #DC3545 — используются для обозначения положительных и отрицательных действий (например, отправка формы, удаление, подтверждение), а также в качестве визуальных маркеров в интерфейсе.

На рисунке 4 представлены основные интерфейсные элементы UI Kit. В верхней части — палитра, структурированная по функциям: тёмные оттенки для фона, средние — для вторичных областей, светлые — для акцентов и контраста. Под палитрой размещены элементы фильтрации. Нижний блок содержит навигационные и функциональные иконки.

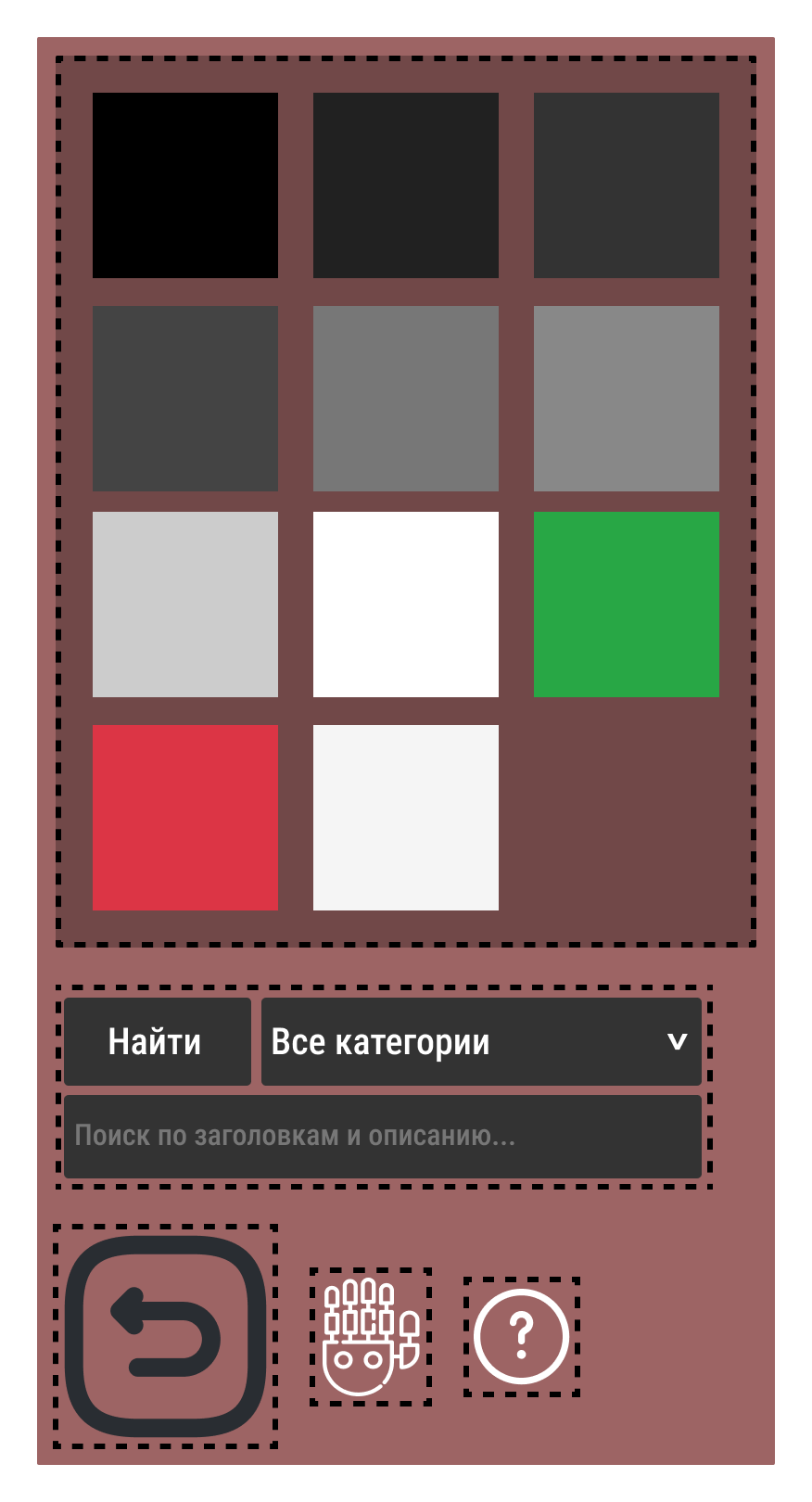


Рисунок 4 - UI Kit

## Повторяющиеся блоки интерфейса

Макет-дизайн веб-приложения «Библиотека ИИ и его эффективное использование» имеет повторяющиеся блоки интерфейсов, большинство из них встречаются на разных страницах и имеют один и тот же вид и то же назначение, например, как блок с навигацией и подвал, эти блоки встречаются на каждой страницы веб-приложения.

Поэтому в этой в подглаве будут представлены повторяющиеся блоки интерфейса:

1. Поиск и фильтрация (см. Рисунок 5):

Блок поиска расположен в верхней части страницы и включает:

* Поле ввода с текстом "Поиск по заголовкам и описанию...";
* Выпадающие списки для фильтрации;
* Кнопку "Найти" для выполнения поиска;

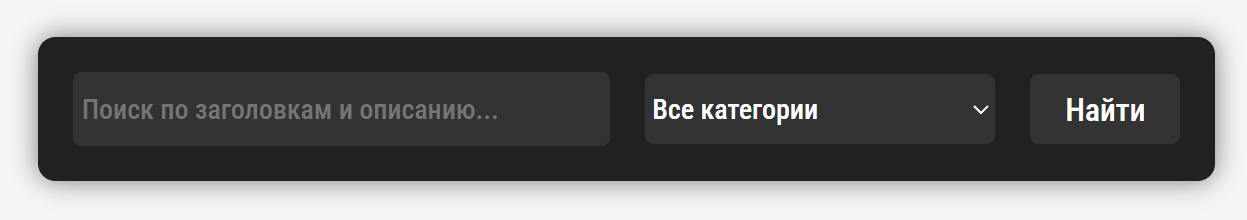


Рисунок 5 – блок «Поиск и фильтрация»

Для авторизованного пользователя дополнительно отображается кнопка "Создать …", с подсказкой, в зависимости в каком разделе он находится (см. Рисунок 6), позволяющая перейти к форме публикации нового материала.

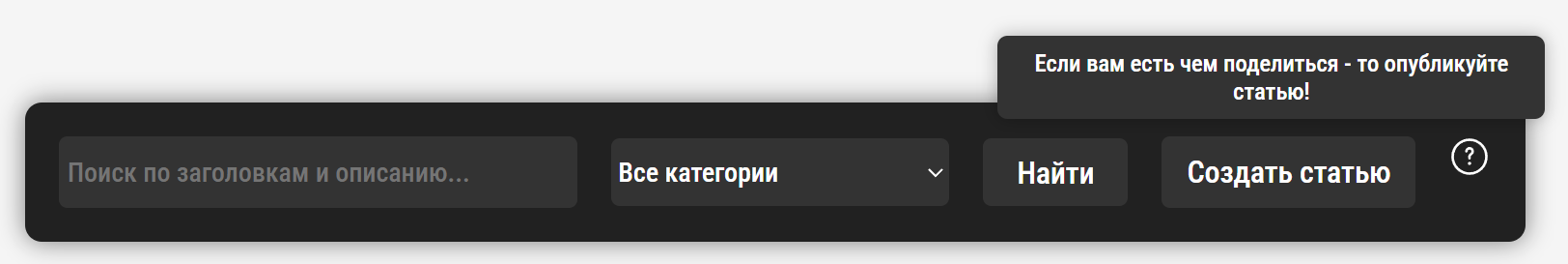


Рисунок 6 – блок «Поиск и фильтрация» авторизированного пользователя

1. Навигация (см. Рисунок 7):

Блок навигации расположен в верхней части страницы и включает:

* Логотип веб приложения;
* Навигационные ссылки на страницы;



Рисунок 7 – блок навигации

1. Подвал (см. Рисунок 8):

Блок навигации расположен в нижней части страницы и включает:

* Логотип веб приложения;
* Некоторая информация для связи;

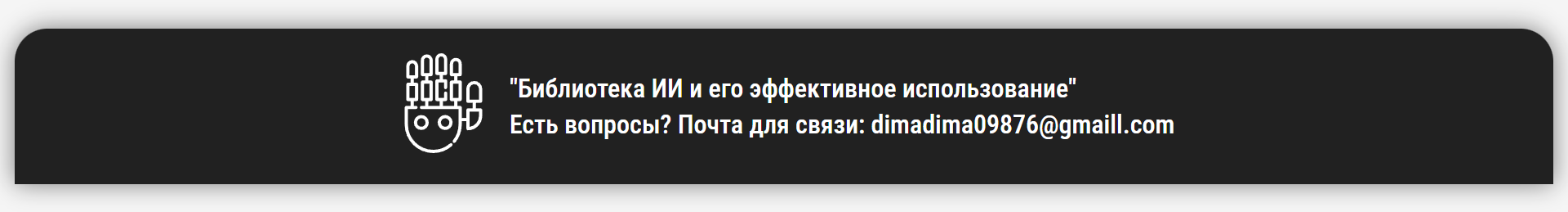


Рисунок 8 – блок подвала

1. Комментарии (см. Рисунок 9):

Блок комментариев расположен в нижней части страницы, после карточки сущности и включает:

* Поле ввода с текстом "Комментарий...";
* Комментарии других пользователей;
* Кнопку "Отправить";

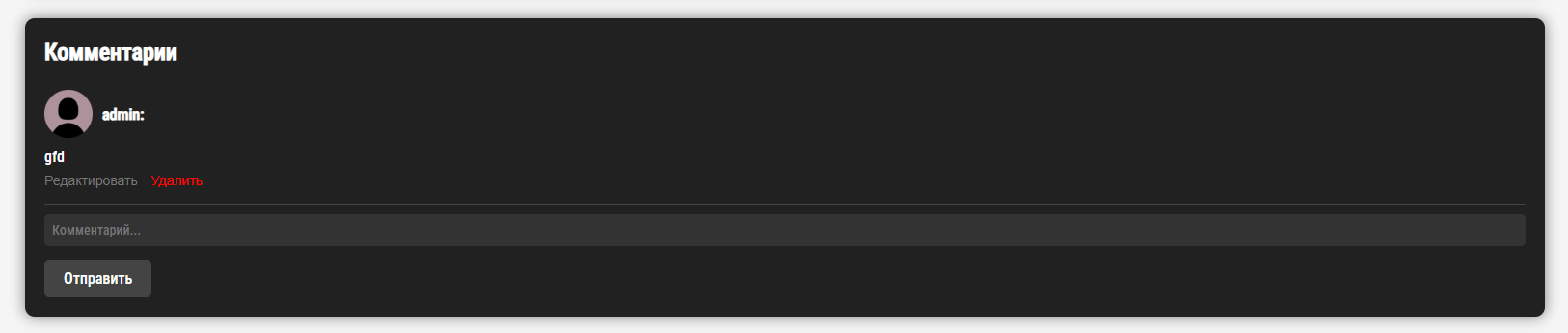


Рисунок 9 – блок комментариев

Для неавторизованного пользователя вместо поля ввода комментария будет отображаться текст: ''Только авторизованные пользователи могут оставлять комментарии'' (см. Рисунок 10).

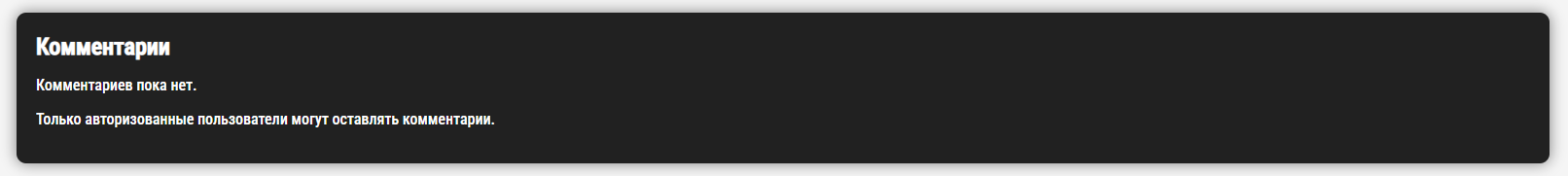


Рисунок 10 – блок комментариев

## Главная страница

На рисунке 5 показана главная страница, т.е. страница статей, если пользователь не авторизовался в системе, то функция создания статьи отсутствует. На экране отображаются карточки опубликованных статей, созданных пользователями. Также на странице присутствует фильтр и поиск по названиям и описанию.

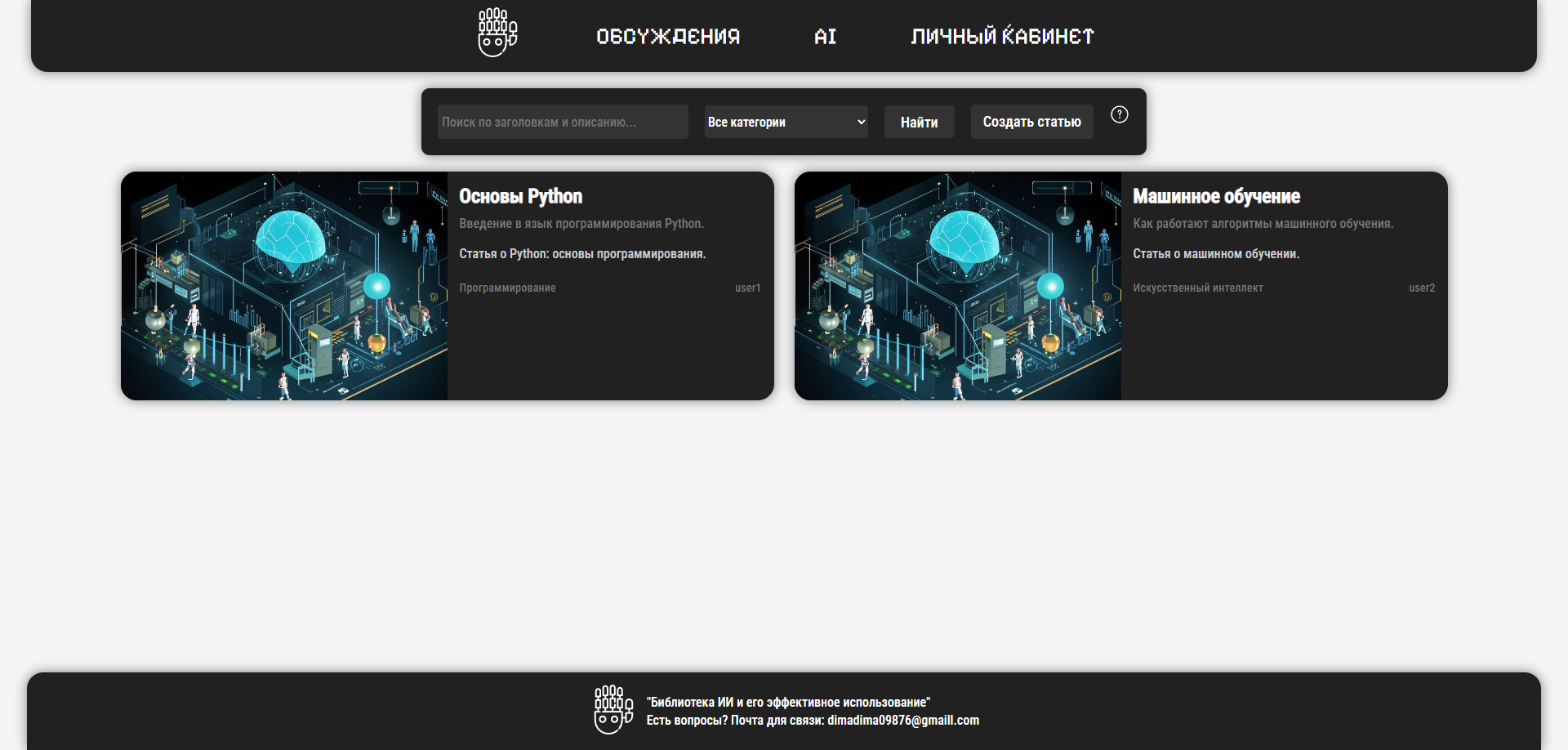


Рисунок 5 – Главная страница

## Страница статьи

На рисунке 6 показана страница статьи, если пользователь создал данную статью, то ему доступна функция редактирования статьи, администраторам всегда доступна это функция. На экране отображается карточка статьи, которая содержит все поля сущности article, а под ней блок с комментариями, если к данной статье отсутствуют комментарии – так и пишет, что комментариев к данной статье ещё нет. И в конце страницы навигационная кнопка «Назад» (Данная кнопка есть на каждой индивидуальной странице).

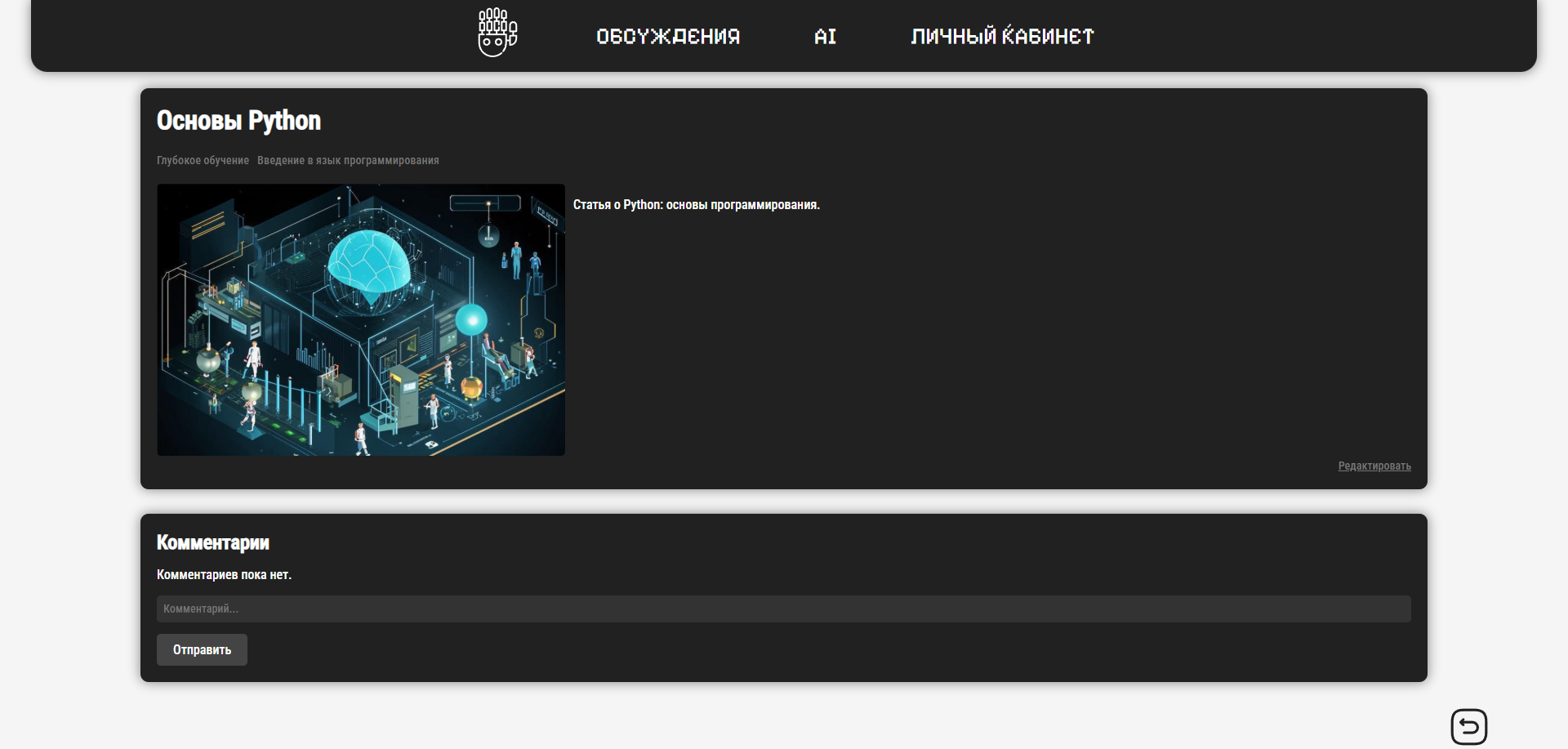


Рисунок 6 – Страница статьи

# **ГЛАВА 4. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ**

## Начальная настройка серверной части

На этапе начальной настройки серверной части веб-приложения, разработанного с использованием фреймворка Laravel, была выполнена конфигурация основных параметров среды, а также подготовка проекта к запуску и корректной работе. Данный этап включал настройку конфигурационного файла .env, генерацию ключа приложения, создание символьной ссылки к хранилищу файлов, а также формирование файла .htaccess для корректной обработки маршрутов.

Файл .env содержит ключевые параметры конфигурации, необходимые для функционирования Laravel-приложения. В процессе настройки были указаны следующие параметры:

* DB\_CONNECTION
* DB\_DATABASE
* SESSION\_DRIVER

После внесения всех значений была выполнена генерация ключа приложения командой (см. Листинг 1):

*Листинг 1 - Команда генерации ключа приложения*

php artisan key:generate

Данная команда создает и сохраняет в .env уникальный криптографический ключ, необходимый для шифрования данных и обеспечения безопасности.

Для организации доступа к пользовательским файлам и другим загружаемым данным была выполнена команда (см. Листинг 2):

*Листинг 2 - Команда генерации символической ссылки*

php artisan storage:link

В результате создается символьная ссылка из директории public/storage в директорию storage/app/public. Это позволяет безопасно и корректно обслуживать файлы, загружаемые через приложение.

Файл .htaccess был создан в корневом каталоге для обеспечения правильной маршрутизации и обработки запросов (см. Листинг 3):

*Листинг 3 - Код файла .htaccess*

RewriteEngine On  
RewriteRule ^(.\*)$ public/$1 [L]

Этот файл обеспечивает, чтобы все запросы, не относящиеся к существующим файлам или директориям, перенаправлялись на index.php, что необходимо для работы маршрутизации Laravel.

Дополнительно в директории public/ была создана пользовательская директория assets, предназначенная для хранения статических ресурсов, используемых на стороне клиента. В рамках структуры assets были организованы следующие вложенные каталоги:

* assets/images – для хранения изображений, используемых в интерфейсе приложения;
* assets/css – для размещения пользовательских каскадных таблиц стилей (CSS);
* assets/js – для хранения JavaScript-файлов, обеспечивающих интерактивность пользовательского интерфейса.

Такая организация позволяет централизованно управлять статическими ресурсами, обеспечивать их удобное подключение в шаблонах и облегчать сопровождение клиентской части проекта.

Для реализации системы аутентификации пользователей в проект был интегрирован пакет **Laravel Sanctum**. Sanctum предоставляет лёгкий способ реализации токен-базированной аутентификации, а также поддержку SPA и API-защиты. Установка и настройка пакета выполнялись с помощью следующей команды (см. Листинг 4):

*Листинг 4 - Команда установки пакета Sanctum*

composer **require** laravel/sanctum

## Генерация основных компонентов серверной логики

На следующем этапе разработки серверной части веб-приложения была произведена генерация ключевых структурных компонентов, реализующих логику системы. С использованием встроенных инструментов командной строки Laravel (Artisan) были созданы необходимые модели, контроллеры, политики доступа и классы запросов.

Модели являются центральными элементами взаимодействия с базой данных. Каждая модель отражает соответствующую сущность предметной области приложения. Выполненные команды для генерации моделей (см. Листинг 5):

*Листинг 5 - Команды создания моделей*

php artisan make:model Article

php artisan make:model ArticleCategory

php artisan make:model ArtificialIntelligence

php artisan make:model Comment

php artisan make:model Discussion

php artisan make:model DiscussionCategory

php artisan make:model Role

php artisan make:model Task

php artisan make:model Transformation

php artisan make:model User

Каждая из указанных моделей будет впоследствии связана с соответствующей таблицей в базе данных и дополнена отношениями и методами для работы с данными.

Контроллеры обеспечивают обработку HTTP-запросов и реализацию прикладной логики. Были сгенерированы следующие контроллеры (см. Листинг 6):

*Листинг 6 - Команды создания контроллеров*

php artisan make:controller AuthController

php artisan make:controller AIController

php artisan make:controller ArticleController

php artisan make:controller CommentController

php artisan make:controller DiscussionController

Каждый контроллер организует соответствующие CRUD-операции, а также взаимодействие с сервисами, моделями и политиками.

Для управления авторизацией действий пользователей над ресурсами были созданы политики доступа (см. Листинг 7):

*Листинг 7 - Команды создания политик*

php artisan make:policy AiPolicy

php artisan make:policy ArticlePolicy

php artisan make:policy CommentPolicy

php artisan make:policy DiscussionPolicy

Эти классы позволяют определить правила, по которым пользователи могут создавать, просматривать, изменять или удалять определенные сущности приложения.

Для централизованной валидации данных, поступающих от пользователя, были созданы специализированные классы запросов. Они позволяют описать правила валидации и авторизации в одном месте, способствуя читаемости и переиспользуемости кода. Генерация осуществлялась следующими командами (см. Листинг 8):

*Листинг 8 - Команды создания реквестов*

php artisan make:request AICreateRequest

php artisan make:request AIUpdateRequest

php artisan make:request ArticleCreateRequest

php artisan make:request ArticleUpdateRequest

php artisan make:request DiscussionCreateRequest

php artisan make:request DiscussionUpdateRequest

php artisan make:request CommentCreateRequest

php artisan make:request CommentUpdateRequest

php artisan make:request RegisterRequest

php artisan make:request LoginRequest

php artisan make:request UpdateProfileRequest

Эти классы используются в соответствующих методах контроллеров и позволяют строго контролировать корректность входящих данных на уровне сервера.

## Маршрутизация

Следующим этапом разработки серверной части стало определение маршрутов, обеспечивающих навигацию и взаимодействие между клиентской и серверной частью веб-приложения. Конфигурация маршрутов осуществлялась в файле routes/web.php. Все маршруты были организованы с учетом стандартов REST и принципов безопасности, таких как разграничение доступа для аутентифицированных и неаутентифицированных пользователей.

Также с целью упрощения обработки CRUD-операций были задействованы ресурсные маршруты, предоставляемые фреймворком Laravel. Они позволяют автоматически создавать стандартные маршруты (index, create, store, show, edit, update, destroy) для соответствующих контроллеров (см. Листинг 9):

*Листинг 9 - Маршруты*

<?php  
use App\Http\Controllers\AIController;  
use App\Http\Controllers\ArticleController;  
use App\Http\Controllers\Auth\AuthController;  
use App\Http\Controllers\CommentController;  
use App\Http\Controllers\DiscussionController;  
use Illuminate\Support\Facades\Route;  
*// Главная страница*Route::get('/', function () {  
 return redirect()->route('articles.index');  
})->name('home');  
*// Ресурсные маршруты*Route::resource('articles', ArticleController::class);  
Route::resource('ai', AIController::class);  
Route::resource('discussions', DiscussionController::class);  
*// Регистрация и авторизация*Route::get('/register', [AuthController::class, 'showRegistrationForm'])->name('register.form')->middleware('guest');  
Route::post('/register', [AuthController::class, 'register'])->name('register')->middleware('guest');  
Route::get('/login', [AuthController::class, 'showLoginForm'])->name('login.form')->middleware('guest');  
Route::post('/login', [AuthController::class, 'login'])->name('login')->middleware('guest');  
*// Защищенные маршруты (требуют аутентификации)*Route::middleware('auth')->group(function () {  
 Route::get('/profile', [AuthController::class, 'show'])->name('profile');  
 Route::put('/profile', [AuthController::class, 'update'])->name('profile.update');  
 Route::post('/logout', [AuthController::class, 'logout'])->name('logout');  
});  
Route::resource('comments', CommentController::class)->middleware('auth');

## Миграции и наполнение базы данных

В рамках данного проекта структура базы данных была предварительно спроектирована на этапе проектирования с использованием веб-платформы **DrawDB**, а затем реализована непосредственно через **SQL-скрипты в phpMyAdmin**.

Для наполнения базы данных начальными тестовыми данными были созданы **seeders** – классы, предназначенные для автоматического заполнения таблиц примерами записей. Это особенно важно для разработки и тестирования функционала без необходимости ручного ввода информации. Сидеры были сгенерированы с помощью команды (см. Листинг 10):

*Листинг 10 - Команда создания сидера*

php artisan make:seeder <SeederName>

В рамках проекта были реализованы сидеры для следующих сущностей:

* UserSeeder – создание тестовых пользователей;
* RoleSeeder – начальные роли;
* TransformationsSeeder и TasksSeeder – создание задачей и преобразований ИИ;
* DiscussionCategoriesSeeder и ArticleCategoriesSeeder – категории обсуждений и статей;
* DiscussionsSeeder и ArticlesSeeder – создание обсуждений и статей;
* CommentsSeeder – комментарии к обсуждениям и статьям;
* ArtificialIntelligencesSeeder – создание тестовых ИИ;
* AiTransformationsSeeder и AiTasksSeeder – связующие таблицы;

После создания сидеров была отредактирована конфигурация класса DatabaseSeeder, где в метод run() были добавлены вызовы всех необходимых сидеров.

В листинге 11 представлен общий обязательный сидер, который находится в файле database/seeders/DatabaseSeeder.php. Этот файл используется для вызова других сидеров и управления процессом заполнения базы данных.

*Листинг 11 - Общий обязательный сидер database/seeders/DatabaseSeeder.php*

class DatabaseSeeder extends Seeder  
{  
 public function run()  
 {  
 $this->call([  
 RolesSeeder::class,  
 UsersSeeder::class,  
 ArticleCategoriesSeeder::class,  
 DiscussionCategoriesSeeder::class,  
 ArticlesSeeder::class,  
 DiscussionsSeeder::class,  
 CommentsSeeder::class,  
 TasksSeeder::class,  
 TransformationsSeeder::class,  
 ArtificialIntelligencesSeeder::class,  
 AiTasksSeeder::class,  
 AiTransformationsSeeder::class,  
 ]);  
 }  
}

Для выполнения заполнения базы данных начальными данными и загрузкой миграции personal\_access\_tokens (Laravel Sanctum) использовалась команда (см. Листинг 11):

*Листинг 12 - Команда миграции с сидерами*

php artisan migrate –seed

## Контроллеры

Для обработки HTTP-запросов я использовал контроллеры — отдельные классы, в которых сосредоточена логика, связанная с определёнными разделами приложения. Это упростило структуру кода и позволило удобно работать с моделями и данными.

Контроллер ArticleController реализует все функции, связанные со статьями (см. Листинг 13): вывод списка, создание, редактирование и удаление. Он обслуживает веб-интерфейс — возвращает представления, а не JSON, как это делают API-контроллеры. Контроллер реализует следующие методы:

1. **index()** — метод, обеспечивающий вывод списка всех статей с возможностью фильтрации по категории и строке поиска. Фильтрация осуществляется с использованием параметров category и search, передаваемых через строку запроса. Дополнительно загружается список всех категорий для отображения в пользовательском интерфейсе. Результат передаётся в шаблон представления articles.index.
2. **show(Article $article)** — отвечает за отображение детальной страницы статьи. Laravel автоматически находит соответствующую запись на основе переданного идентификатора (Model Binding), после чего данные передаются в шаблон articles.show.
3. **create()** — формирует страницу создания новой статьи. Загружается список категорий, необходимых для выбора в форме, и передаётся в шаблон articles.create.
4. **store(ArticleCreateRequest $request)** — обрабатывает POST-запрос на добавление новой статьи. Используется кастомный валидатор ArticleCreateRequest, после чего данные сохраняются в БД. Если загружено изображение-превью, оно сохраняется в директорию storage/public/previews, а путь к нему сохраняется в базе. Также статье автоматически назначается автор — текущий аутентифицированный пользователь. По завершении выполняется редирект с сообщением об успехе.
5. **edit(Article $article)** — отображает форму редактирования существующей статьи, включая возможность смены категории и замены изображения-превью. Данные статьи и список категорий передаются в представление articles.edit.
6. **update(ArticleUpdateRequest $request, Article $article)** — обновляет информацию о статье. В случае, если пользователь загрузил новое изображение, предыдущее удаляется из хранилища, а новое сохраняется с обновлением соответствующего поля. Валидированные данные применяются к текущей записи в базе. После успешного обновления пользователь перенаправляется на страницу просмотра статьи.
7. **destroy(Article $article)** — удаляет статью из базы данных. Если у статьи имеется привязанное изображение-превью, оно предварительно удаляется с диска. После удаления происходит перенаправление на список статей с соответствующим уведомлением.

Контроллер использует механизм авторизации ресурсов Laravel через метод authorizeResource, что позволяет централизованно применять политику ArticlePolicy к каждому из методов (view, update, delete и др.), обеспечивая контроль доступа к операциям на уровне кода.

*Листинг 13 - Контроллер статей app/Http/Controllers/ArticleController.php*

<?php  
namespace App\Http\Controllers;  
use App\Http\Requests\ArticleCreateRequest;  
use App\Http\Requests\ArticleUpdateRequest;  
use App\Models\Article;  
use App\Models\ArticleCategory;  
use Illuminate\Support\Facades\Auth;  
use Illuminate\Support\Facades\Storage;  
class ArticleController extends Controller  
{  
 public function \_\_construct()  
 {  
 *// Автоматически применяет политику (ArticlePolicy) ко всем методам контроллера  
 // Например, для update() будет вызван метод update() в политике* $this->authorizeResource(Article::class, 'article');  
 }  
 *// Список статей с возможностью фильтрации и поиска* public function index()  
 {  
 *// Получаем параметры поиска и фильтра из GET-запроса* $search = request('search'); *// Поисковая строка* $categoryId = request('category'); *// ID категории для фильтрации  
 // Все категории для выпадающего списка фильтра* $categories = ArticleCategory::*all*();  
 *// Базовый запрос для получения статей* $query = Article::*query*();  
 *// Фильтр по категории (если выбрана)* if ($categoryId) {  
 $query->where('article\_category\_id', $categoryId);  
 }  
 *// Поиск по заголовку или описанию (если задан)* if ($search) {  
 $query->where(function($q) use ($search) {  
 $q->where('title', 'like', "%{$search}%")  
 ->orWhere('description', 'like', "%{$search}%");  
 });  
 }  
 $articles = $query->get();  
 return view('articles.index', compact('articles', 'categories'));  
 }  
 *// Просмотр одной статьи* public function show(Article $article)  
 {  
 *// Laravel автоматически найдет статью по ID (Model Binding)* return view('articles.show', compact('article'));  
 }  
 *// Форма создания новой статьи* public function create()  
 {  
 *// Получаем все категории для выпадающего списка* $articleCategories = ArticleCategory::*all*();  
 return view('articles.create', compact('articleCategories'));  
 }  
 *// Сохранение новой статьи* public function store(ArticleCreateRequest $request)  
 {  
 *// Валидация данных выполняется в ArticleCreateRequest* $validatedData = $request->validated();  
 *// Привязываем статью к текущему пользователю* $validatedData['author\_id'] = Auth::id();  
 *// Если загружено изображение, сохраняем его в storage/public/previews* if ($request->hasFile('preview')) {  
 $validatedData['preview'] = $request->file('preview')->store('previews', 'public');  
 }  
 *// Создаем статью и перенаправляем на список статей* Article::*create*($validatedData);  
 return redirect()->route('articles.index')->with('success', 'Статья успешно создана!');  
 }  
 *// Форма редактирования статьи* public function edit(Article $article)  
 {  
 $articleCategories = ArticleCategory::*all*();  
 return view('articles.edit', compact('article', 'articleCategories'));  
 }  
 *// Обновление статьи* public function update(ArticleUpdateRequest $request, Article $article)  
 {  
 $validated = $request->validated();  
 *// Если загружено новое изображение* if ($request->hasFile('preview')) {  
 *// Удаляем старое изображение (если оно было)* if ($article->preview) {  
 Storage::disk('public')->delete($article->preview);  
 }  
 *// Сохраняем новое изображение* $validated['preview'] = $request->file('preview')->store('previews', 'public');  
 }  
 $article->update($validated);  
 return redirect()->route('articles.show', $article)->with('success', 'Статья успешно обновлена!');  
 }  
 *// Удаление статьи* public function destroy(Article $article)  
 {  
 *// Если у статьи есть превью, удаляем файл из хранилища* if ($article->preview) {  
 Storage::disk('public')->delete($article->preview);  
 }  
 *// Удаляем статью из БД* $article->delete();  
 return redirect()->route('articles.index')->with('success', 'Статья успешно удалена!');  
 }  
}

Контроллер AIController реализует логику управления сущностью искусственного интеллекта (ИИ) (см. Листинг 14). Используется встроенный механизм авторизации authorizeResource, обеспечивающий применение политики AiPolicy ко всем действиям контроллера. Ниже приведено описание каждого метода:

1. **index()** — метод, обеспечивающий вывод списка всех ИИ с возможностью фильтрации по задачам (task) и трансформациям (transformation), а также по строке поиска (search). Для фильтрации используется отношение many-to-many. Загружаются все доступные задачи и трансформации, которые передаются в шаблон ai.index вместе с результатом выборки.
2. **show(ArtificialIntelligence $ai)** — отображает страницу с детальной информацией об отдельном ИИ. Благодаря механизму Model Binding нужная запись автоматически передаётся в шаблон ai.show.
3. **create()** — формирует страницу для создания нового ИИ. Загружаются все задачи и трансформации, которые пользователь может выбрать при заполнении формы. Данные передаются в представление ai.create.
4. **store(AICreateRequest $request)** — обрабатывает POST-запрос на добавление нового ИИ. Используется валидатор AICreateRequest. После создания основной записи связывает ИИ с выбранными задачами и трансформациями через соответствующие pivot-таблицы. По завершении выполняется редирект на список ИИ с сообщением об успешном добавлении.
5. **edit(ArtificialIntelligence $ai)** — отображает форму редактирования существующего ИИ. Загружаются данные текущей записи, а также все возможные задачи и трансформации. Информация передаётся в шаблон ai.edit.
6. **update(AIUpdateRequest $request, ArtificialIntelligence $ai)** — обновляет данные ИИ. Сначала происходит обновление основной информации, затем синхронизируются связи many-to-many: если какие-то ID не переданы, соответствующие связи удаляются. После успешного обновления пользователь перенаправляется на список ИИ с уведомлением.
7. **destroy(ArtificialIntelligence $ai)** — удаляет выбранную запись ИИ. Связанные данные удаляются автоматически благодаря ORM. После удаления выполняется редирект на страницу со списком ИИ и сообщением об успешной операции.

*Листинг 14 - Контроллер ИИ app/Http/Controllers/AIController.php*

<?php  
namespace App\Http\Controllers;  
use App\Http\Requests\AICreateRequest;  
use App\Http\Requests\AIUpdateRequest;  
use App\Models\ArtificialIntelligence;  
use App\Models\Task;  
use App\Models\Transformation;  
  
class AIController extends Controller  
{  
 public function \_\_construct()  
 {  
 *// Автоматически применяет политику (AiPolicy) ко всем методам  
 // Проверка прав происходит через методы view, create, update, delete в политике* $this->authorizeResource(ArtificialIntelligence::class, 'ai');  
 }  
 *//Список ИИ с фильтрацией по задачам и трансформациям* public function index()  
 {  
 *// Параметры фильтрации из GET-запроса* $search = request('search');  
 $transformationId = request('transformation');  
 $taskId = request('task');  
 *// Все доступные трансформации и задачи для фильтров* $transformations = Transformation::*all*();  
 $tasks = Task::*all*();  
 *// Базовый запрос с жадной загрузкой связей* $query = ArtificialIntelligence::*with*(['transformations', 'tasks']);  
 *// Фильтр по трансформациям (через отношение many-to-many)* if ($transformationId) {  
 $query->whereHas('transformations', function ($q) use ($transformationId) {  
 $q->where('transformations.id', $transformationId);  
 });  
 }  
 *// Фильтр по задачам (через отношение many-to-many)* if ($taskId) {  
 $query->whereHas('tasks', function ($q) use ($taskId) {  
 $q->where('tasks.id', $taskId);  
 });  
 }  
 *// Поиск по названию или описанию* if ($search) {  
 $query->where(function ($q) use ($search) {  
 $q->where('name', 'like', "%{$search}%")  
 ->orWhere('description', 'like', "%{$search}%");  
 });  
 }  
 $ais = $query->get();  
 return view('ai.index', compact('ais', 'transformations', 'tasks'));  
 }  
 *//Просмотр конкретного ИИ* public function show(ArtificialIntelligence $ai)  
 {  
 *// Автоматическая загрузка через Model Binding* return view('ai.show', compact('ai'));  
 }  
 *//Форма создания нового ИИ* public function create()  
 {  
 *// Все задачи и трансформации для выбора при создании* $tasks = Task::*all*();  
 $transformations = Transformation::*all*();  
 return view('ai.create', compact('tasks', 'transformations'));  
 }  
 *//Сохранение нового ИИ* public function store(AICreateRequest $request)  
 {  
 *// Создаем основную запись ИИ (исключаем поля связей)* $ai = ArtificialIntelligence::*create*($request->except(['task\_ids', 'transformation\_ids']));  
 *// Привязываем выбранные задачи (many-to-many)* if ($request->has('task\_ids')) {  
 $ai->tasks()->attach($request->task\_ids);  
 }  
 *// Привязываем выбранные трансформации (many-to-many)* if ($request->has('transformation\_ids')) {  
 $ai->transformations()->attach($request->transformation\_ids);  
 }  
 return redirect()->route('ai.index')->with('success', 'ИИ успешно добавлен!');  
 }  
 *//Форма редактирования ИИ* public function edit(ArtificialIntelligence $ai)  
 {  
 *// Проверка прав через политику (автоматически через authorizeResource)* $tasks = Task::*all*();  
 $transformations = Transformation::*all*();  
 return view('ai.edit', compact('ai', 'tasks', 'transformations'));  
 }  
 *//Обновление данных ИИ* public function update(AIUpdateRequest $request, ArtificialIntelligence $ai)  
 {  
 *// Обновляем основную информацию* $ai->update($request->validated());  
 *// Синхронизируем связанные задачи (many-to-many)  
 // Если task\_ids не передано - используем пустой массив (удаляем все связи)* $ai->tasks()->sync($request->input('task\_ids', []));  
 *// Синхронизируем связанные трансформации (many-to-many)* $ai->transformations()->sync($request->input('transformation\_ids', []));  
 return redirect()->route('ai.index')->with('success', 'ИИ обновлён!');  
 }  
  
 *//Удаление ИИ* public function destroy(ArtificialIntelligence $ai)  
 {  
 *// Удаление автоматически проверяется через политику (метод delete)* $ai->delete();  
 return redirect()->route('ai.index')->with('success', 'ИИ удалён!');  
 }  
}

Контроллер CommentController реализует базовую CRUD-логику для управления комментариями (см. Листинг 15). Используется метод authorizeResource, благодаря которому ко всем действиям автоматически применяется политика CommentPolicy, обеспечивая контроль доступа. Описание методов:

1. **store(CommentCreateRequest $request)** — обрабатывает создание нового комментария. Используется валидатор CommentCreateRequest, дополнительно к валидированным данным добавляется ID текущего пользователя. Комментарий сохраняется в базе. После сохранения пользователь перенаправляется на страницу статьи или обсуждения, в зависимости от привязки комментария. Если привязка отсутствует, возвращается с ошибкой.
2. **edit(Comment $comment)** — используется для активации формы редактирования на клиентской стороне. Возвращает пользователя обратно с флеш-сообщением edit\_comment\_id, содержащим ID комментария, который нужно отобразить для редактирования.
3. **update(CommentUpdateRequest $request, Comment $comment)** — обновляет текст комментария. Валидатор CommentUpdateRequest проверяет данные, после чего производится обновление только поля text. Пользователь перенаправляется на страницу привязанной статьи или обсуждения с уведомлением об успешном обновлении.
4. **destroy(Comment $comment)** — удаляет указанный комментарий. Перед удалением сохраняются ID связанной статьи или обсуждения. После удаления выполняется редирект обратно на соответствующую страницу с сообщением об успехе.

*Листинг 15 – Контроллер комментариев app/Http/Controllers/CommentController.php*

<?php  
namespace App\Http\Controllers;  
use App\Http\Requests\CommentCreateRequest;  
use App\Http\Requests\CommentUpdateRequest;  
use App\Models\Comment;  
use Illuminate\Support\Facades\Auth;  
  
class CommentController extends Controller  
{  
 public function \_\_construct()  
 {  
 *// Автоматически применяет политику (CommentPolicy) ко всем методам* $this->authorizeResource(Comment::class, 'comment');  
 }  
 *//Создание нового комментария* public function store(CommentCreateRequest $request)  
 {  
 *// Получаем валидированные данные и добавляем ID текущего пользователя* $data = $request->validated();  
 $data['user\_id'] = Auth::id();  
 *// Создаем комментарий* $comment = Comment::*create*($data);  
 *// Перенаправляем обратно с сообщением в зависимости от типа комментария* if ($request->filled('article\_id')) {  
 return redirect()->route('articles.show', $request->input('article\_id'))  
 ->with('success', 'Комментарий добавлен!');  
 }  
 if ($request->filled('discussion\_id')) {  
 return redirect()->route('discussions.show', $request->input('discussion\_id'))  
 ->with('success', 'Комментарий добавлен!');  
 }  
 return back()->withErrors('Комментарий не привязан ни к статье, ни к обсуждению.');  
 }  
 *//Редактирование комментария (возвращает ID для фронтенда)* public function edit(Comment $comment)  
 {  
 *// Проверка прав выполняется автоматически через authorizeResource()* return back()->with('edit\_comment\_id', $comment->id);  
 }  
 *//Обновление комментария* public function update(CommentUpdateRequest $request, Comment $comment)  
 {  
 *// Проверка прав выполняется автоматически через authorizeResource()* $validated = $request->validated();  
 *// Обновляем только текст комментария* $comment->update(['text' => $validated['text']]);  
 *// Перенаправляем обратно с сообщением в зависимости от типа комментария* if ($comment->article\_id) {  
 return redirect()->route('articles.show', $comment->article\_id)  
 ->with('success', 'Комментарий обновлён!');  
 }  
 if ($comment->discussion\_id) {  
 return redirect()->route('discussions.show', $comment->discussion\_id)  
 ->with('success', 'Комментарий обновлён!');  
 }  
 return back()->with('success', 'Комментарий обновлён!');  
 }  
 *//Удаление комментария* public function destroy(Comment $comment)  
 {  
 *// Сохраняем ID связанных сущностей перед удалением* $articleId = $comment->article\_id;  
 $discussionId = $comment->discussion\_id;  
 *// Удаляем комментарий* $comment->delete();  
 *// Перенаправляем обратно с сообщением в зависимости от типа комментария* if ($articleId) {  
 return redirect()->route('articles.show', $articleId)  
 ->with('success', 'Комментарий удалён!');  
 }  
 if ($discussionId) {  
 return redirect()->route('discussions.show', $discussionId)  
 ->with('success', 'Комментарий удалён!');  
 }  
 return back()->with('success', 'Комментарий удалён!');  
 }  
}

Контроллер DiscussionController управляет логикой создания, редактирования, отображения и удаления обсуждений (см. Листинг 16). Через метод authorizeResource ко всем действиям применяется политика DiscussionPolicy, обеспечивающая контроль доступа. Описание методов:

1. **index()** — выводит список обсуждений с возможностью фильтрации по категории (category) и поиску по заголовку или описанию (search). Загружаются все доступные категории для отображения в фильтре. Результат, отсортированный по дате создания, передаётся в шаблон discussions.index.
2. **create()** — отображает форму создания нового обсуждения. Загружаются все категории, которые можно выбрать в форме. Данные передаются в шаблон discussions.create.
3. **store(DiscussionCreateRequest $request)** — обрабатывает создание нового обсуждения. Валидированные данные дополняются ID текущего пользователя (author\_id). При наличии загруженного изображения-превью оно сохраняется в директорию storage/public/previews, а путь записывается в базу. После создания обсуждение добавляется в базу, и пользователь перенаправляется на список обсуждений с уведомлением об успехе.
4. **show(Discussion $discussion)** — отображает страницу конкретного обсуждения. Laravel автоматически подставляет модель по ID через механизм Model Binding. Данные передаются в шаблон discussions.show.
5. **edit(Discussion $discussion)** — используется для активации редактирования на фронтенде. Возвращает пользователя обратно с флеш-сообщением edit\_discussion\_id, содержащим ID редактируемого обсуждения.
6. **update(DiscussionUpdateRequest $request, Discussion $discussion)** — обновляет информацию об обсуждении. При загрузке нового изображения-превью предыдущее удаляется из хранилища, а новое сохраняется и путь обновляется. Все валидированные данные применяются к существующей записи. По завершении — редирект с сообщением об успешном обновлении.
7. **destroy(Discussion $discussion)** — удаляет обсуждение из базы. Если к обсуждению было привязано изображение, оно удаляется вместе с записью. После этого выполняется перенаправление на список обсуждений с уведомлением об удалении.

*Листинг 16 - Контроллер обсуждений app/Http/Controllers/DiscussionController.php*

<?php  
namespace App\Http\Controllers;  
use App\Http\Requests\DiscussionCreateRequest;  
use App\Http\Requests\DiscussionUpdateRequest;  
use App\Models\Discussion;  
use App\Models\DiscussionCategory;  
use Illuminate\Support\Facades\Auth;  
use Illuminate\Support\Facades\Storage;  
  
class DiscussionController extends Controller  
{  
 public function \_\_construct()  
 {  
 *// Автоматически применяет политику (DiscussionPolicy) ко всем методам* $this->authorizeResource(Discussion::class, 'discussion');  
 }  
 *//Отображение списка обсуждений с возможностью фильтрации* public function index()  
 {  
 *// Получаем параметры фильтрации из GET-запроса* $search = request('search');  
 $categoryId = request('category');  
 *// Загружаем все категории для выпадающего списка фильтров* $categories = DiscussionCategory::*all*();  
 *// Базовый запрос с сортировкой по умолчанию (можно добавить ->latest())* $query = Discussion::*query*();  
 *// Фильтр по категории* if ($categoryId) {  
 *// Исправлено: было 'article\_category\_id', должно быть 'discussion\_category\_id'* $query->where('discussion\_category\_id', $categoryId);  
 }  
 *// Поиск по заголовку или описанию* if ($search) {  
 $query->where(function($q) use ($search) {  
 $q->where('title', 'like', "%{$search}%")  
 ->orWhere('description', 'like', "%{$search}%");  
 });  
 }  
 *// Получаем отфильтрованный результат* $discussions = $query->get();  
 return view('discussions.index', compact('discussions', 'categories'));  
 }  
 *//Форма создания нового обсуждения* public function create()  
 {  
 $categories = DiscussionCategory::*all*();  
 return view('discussions.create', compact('categories'));  
 }  
 *//Сохранение нового обсуждения* public function store(DiscussionCreateRequest $request)  
 {  
 $data = $request->validated();  
 *// Обработка загрузки превью* if ($request->hasFile('preview')) {  
 $data['preview'] = $request->file('preview')->store('previews', 'public');  
 }  
 *// Привязываем обсуждение к текущему пользователю* $data['author\_id'] = Auth::id();  
 *// Создаем обсуждение* Discussion::*create*($data);  
 return redirect()->route('discussions.index')->with('success', 'Обсуждение создано!');  
 }  
 *//Просмотр конкретного обсуждения* public function show(Discussion $discussion)  
 {  
 return view('discussions.show', compact('discussion'));  
 }  
 *//Форма редактирования обсуждения* public function edit(Discussion $discussion)  
 {  
 *// Проверка прав выполняется автоматически через authorizeResource()* return back()->with('edit\_discussion\_id', $discussion->id);  
 }  
 *//Обновление обсуждения* public function update(DiscussionUpdateRequest $request, Discussion $discussion)  
 {  
 $data = $request->validated();  
 *// Обработка обновления превью* if ($request->hasFile('preview')) {  
 *// Удаляем старое изображение, если оно есть* if ($discussion->preview) {  
 Storage::disk('public')->delete($discussion->preview);  
 }  
 *// Сохраняем новое изображение* $data['preview'] = $request->file('preview')->store('previews', 'public');  
 }  
 *// Обновляем обсуждение* $discussion->update($data);  
 return redirect()->route('discussions.index')->with('success', 'Обсуждение успешно обновлено!');  
 }  
 *//Удаление обсуждения* public function destroy(Discussion $discussion)  
 {  
 $discussion->delete();  
 return redirect()->route('discussions.index')->with('success', 'Обсуждение удалено.');  
 }  
}

Контроллер AuthController реализует функциональность аутентификации, регистрации, обновления профиля и выхода из системы (см. Листинг 17). Обеспечивает взаимодействие с формами входа и регистрации, а также управление пользовательским профилем. Описание методов:

1. **showRegistrationForm()** — отображает форму регистрации нового пользователя. Данные передаются в шаблон auth.register.
2. **register(RegisterRequest $request)** — обрабатывает отправку формы регистрации. Валидированные данные дополняются: ID роли по умолчанию (user) и путь к загруженному аватару, если он предоставлен. Затем создаётся пользователь, выполняется автоматический вход в систему, генерируется токен и происходит редирект на главную страницу с приветственным сообщением.
3. **showLoginForm()** — отображает форму входа пользователя. Данные передаются в шаблон auth.login.
4. **login(LoginRequest $request)** — обрабатывает вход по email и паролю. В случае успешной аутентификации создается токен, и пользователь перенаправляется на главную страницу с сообщением об успешном входе. В случае ошибки — возврат на форму с сообщением о неверном пароле.
5. **show()** — отображает профиль текущего авторизованного пользователя, включая его обсуждения. Если пользователь не авторизован, перенаправляется на страницу входа. Данные передаются в шаблон auth.profile.
6. **logout(Request $request)** — завершает текущую сессию пользователя и перенаправляет его на главную страницу с сообщением о выходе из аккаунта.
7. **update(UpdateProfileRequest $request)** — обновляет профиль текущего пользователя. Если загружен новый аватар, старый файл удаляется из хранилища, и сохраняется новый путь. При наличии нового пароля он хешируется. Остальные валидированные данные обновляют профиль. После обновления происходит возврат на текущую страницу с сообщением об успешном обновлении.

*Листинг 17 - Контроллер аутентификации app/Http/Controllers/AuthController.php*

<?php  
namespace App\Http\Controllers\Auth;  
use App\Http\Controllers\Controller;  
use App\Http\Requests\LoginRequest;  
use App\Http\Requests\RegisterRequest;  
use App\Http\Requests\UpdateProfileRequest;  
use App\Models\Role;  
use App\Models\User;  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Auth;  
use Illuminate\Support\Facades\Hash;  
use Illuminate\Support\Facades\Storage;  
class AuthController extends Controller  
{  
 public function showRegistrationForm() {  
 return view('auth.register');  
 }  
 public function register(RegisterRequest $request)  
 {  
 *// Находим роль пользователя* $role\_user = Role::*where*('code', 'user')->first();  
 *// Проверяем, загружено ли изображение* $path = null;  
 if ($request->hasFile('avatar\_url')) {  
 $path = $request->file('avatar\_url')->store('avatars', 'public');  
 }  
 *// Создаем нового пользователя* $user = User::*create*([  
 ...$request->validated(),  
 'avatar\_url' => $path,  
 'role\_id' => $role\_user->id,  
 ]);  
 *// Аутентификация* Auth::login($user);  
 *// Создание токена* $user->createToken('default')->plainTextToken;  
 return redirect()->route('home')->with('success', 'Добро пожаловать новый пользователь!');  
 }  
 *// Показать форму входа* public function showLoginForm()  
 {  
 return view('auth.login');  
 }  
 *// Аутентификация* public function login(LoginRequest $request)  
 {  
 $credentials = $request->only('email', 'password');  
 if (!Auth::attempt($credentials)) {  
 return back()  
 ->withErrors(['password' => 'Неверный пароль'])  
 ->withInput($request->only('email'));  
 }  
 $user = Auth::user();  
 $token = $user->createToken('default')->plainTextToken;  
 return redirect()->route('home')->with('token', $token)->with('success', 'Вы успешно вошли в аккаунт!');  
 }  
 public function show()  
 {  
 if (Auth::check()) {  
 $user = Auth::user();  
 $discussions = $user->discussions; *// или Discussion::where('author\_id', $user->id)->get();* return view('auth.profile', compact('discussions'));  
 }  
 return redirect()->route('login');  
 }  
 *// Выход* public function logout(Request $request) {  
 Auth::logout();  
 return redirect()->route('home')->with('success', 'Вы вышли из аккаунта');  
 }  
 public function update(UpdateProfileRequest $request)  
 {  
 $user = Auth::user();  
 *// Если загружен новый аватар* if ($request->hasFile('avatar\_url')) {  
 *// Удаляем старый аватар, если он существует* if ($user->avatar\_url) {  
 Storage::disk('public')->delete($user->avatar\_url);  
 }  
 *// Загружаем новый аватар* $path = $request->file('avatar\_url')->store('avatars', 'public');  
 *// Обновляем поле avatar\_url с новым путём* $user->avatar\_url = $path;  
 }  
 *// Если введен новый пароль, то обновляем его* if ($request->filled('password')) {  
 $user->password = Hash::make($request->password); *// Хешируем новый пароль* }  
 *// Обновляем остальные поля пользователя* $user->update($request->except(['password', 'avatar\_url']));  
 return redirect()->back()->with('success', 'Профиль успешно обновлен');  
 }  
}

## Модели

В Laravel модели применяются для взаимодействия с базой данных посредством Eloquent ORM. Каждая модель представляет собой класс, связанный с определённой таблицей в БД, и служит для управления данными, определения связей между таблицами и реализации бизнес-логики приложения. Благодаря моделям операции чтения, записи и обновления данных становятся более простыми и интуитивно понятными за счёт использования методов и свойств объектно-ориентированного подхода.

В проекте были реализованы следующие ключевые модели:

* Article;
* ArticleCategory;
* ArtificialIntelligence;
* Comment;
* Discussion;
* DiscussionCategory;
* Role;
* Task;
* Transformation;
* User;

**Article**

Модель Article представляет статью (см. Листинг 18), которая содержит некоторые поля. Для установки связей с другими таблицами используются методы articleCategory() (связь «принадлежит» с ArticleCategory) и author() (связь «принадлежит» с User). Также реализована связь один ко многим с моделью Comment.

*Листинг 18 – Модель Article app/Http/Models/Article.php*

class Article extends Model  
{  
 public $timestamps = false; *// Отключаем временные метки* protected $fillable = ['text', 'title', 'description', 'preview', 'article\_category\_id', 'author\_id'];  
 *// Связь с категорией статьи* public function articleCategory()  
 {  
 return $this->belongsTo(ArticleCategory::class, 'article\_category\_id');  
 }  
 *// Связь с автором (пользователем)* public function author()  
 {  
 return $this->belongsTo(User::class, 'author\_id');  
 }  
 *// Связь с комментариями* public function comments()  
 {  
 return $this->hasMany(Comment::class, 'article\_id');  
 }  
}

**ArticleCategory**

Модель ArticleCategory используется для группировки статей по категориям (см. Листинг 19). Устанавливает обратную связь один ко многим с моделью Article.

*Листинг 19 – Модель ArticleCategory app/Http/Models/ArticleCategory.php*

class ArticleCategory extends Model  
{  
 protected $fillable = [  
 'name', 'code',  
 ];  
 *// Связь с статьями* public function articles()  
 {  
 return $this->hasMany(Article::class, 'article\_category\_id');  
 }  
}

**ArtificialIntelligence**

Модель ArtificialIntelligence описывает искусственный интеллект (см. Листинг 20). Реализуются связи многие ко многим через промежуточные таблицы с моделями Task и Transformation.

*Листинг 20 – Модель ArtificialIntelligence app/Http/Models/ArtificialIntelligence.php*

class ArtificialIntelligence extends Model  
{  
 public $timestamps = false;  
 protected $fillable = [  
 'name', 'paid', 'trial', 'link', 'description'  
 ];  
 public function tasks()  
 {  
 return $this->belongsToMany(Task::class, 'ai\_tasks', 'ai\_id', 'task\_id');  
 }  
 public function transformations()  
 {  
 return $this->belongsToMany(Transformation::class, 'ai\_transformations', 'ai\_id', 'transformation\_id');  
 }  
}

**Comment**

Модель Comment представляет комментарий, написанный пользователем (см. Листинг 21). Комментарий может быть прикреплён как к статье (article\_id), так и к обсуждению (discussion\_id). Установлены связи «принадлежит» с моделями User, Article и Discussion.

*Листинг 21 – Модель Comment app/Http/Models/Comment.php*

class Comment extends Model  
{  
 public $timestamps = false; *// Отключаем временные метки* protected $fillable = [  
 'text', 'user\_id', 'discussion\_id', 'article\_id',  
 ];  
 *// Связь с пользователем* public function user()  
 {  
 return $this->belongsTo(User::class, 'user\_id');  
 }  
 *// Связь с обсуждением* public function discussion()  
 {  
 return $this->belongsTo(Discussion::class, 'discussion\_id')->nullable();  
 }  
 *// Связь со статьей* public function article()  
 {  
 return $this->belongsTo(Article::class, 'article\_id')->nullable();  
 }  
}

**Discussion**

Модель Discussion (см. Листинг 22). Поддерживает связь один ко многим с моделью Comment.

*Листинг 22 – Модель Discussion app/Http/Models/Discussion.php*

class Discussion extends Model  
{  
 public $timestamps = false; *// Отключаем временные метки* protected $fillable = [  
 'text', 'title', 'description', 'preview', 'status', 'discussion\_category\_id', 'author\_id',  
 ];  
 *// Связь с категорией обсуждения* public function discussionCategory()  
 {  
 return $this->belongsTo(DiscussionCategory::class, 'discussion\_category\_id');  
 }  
 *// Связь с автором (пользователем)* public function author()  
 {  
 return $this->belongsTo(User::class, 'author\_id');  
 }  
 *// Связь с комментариями* public function comments()  
 {  
 return $this->hasMany(Comment::class, 'discussion\_id');  
 }  
}

**DiscussionCategory**

Модель DiscussionCategory служит для классификации обсуждений по категориям (см. Листинг 23). Устанавливает связь один ко многим с моделью Discussion.

*Листинг 23 – Модель DiscussionCategory app/Http/Models/DiscussionCategory.php*

class DiscussionCategory extends Model  
{  
 protected $fillable = [  
 'name', 'code',  
 ];  
 *// Связь с обсуждениями* public function discussions()  
 {  
 return $this->hasMany(Discussion::class, 'discussion\_category\_id');  
 }  
}

**Role**

Модель Role представляет роль пользователя в системе (см. Листинг 24). Поддерживает связь один ко многим с моделью User.

*Листинг 24 – Модель Role app/Http/Models/Role.php*

class Role extends Model  
{  
 protected $fillable = [  
 'name', 'code',  
 ];  
 *// Связь с пользователями* public function users()  
 {  
 return $this->hasMany(User::class);  
 }  
}

**Task**

Модель Task описывает задачи, которые могут решаться с помощью искусственного интеллекта (см. Листинг 25). Связывается с моделью ArtificialIntelligence через промежуточную таблицу ai\_tasks.

*Листинг 25 – Модель Task app/Http/Models/Task.php*

class Task extends Model  
{  
 protected $fillable = [  
 'name', 'code',  
 ];  
 *// Связь с искусственным интеллектом через промежуточную таблицу ai\_tasks* public function artificialIntelligences()  
 {  
 return $this->belongsToMany(ArtificialIntelligence::class, 'ai\_tasks', 'task\_id', 'ai\_id');  
 }  
}

**Transformation**

Модель Transformation представляет преобразования, применяемые ИИ (см. Листинг 26). Также участвует в связи многие ко многим с моделью ArtificialIntelligence через таблицу ai\_transformations.

*Листинг 26 – Модель Transformation app/Http/Models/Transformation.php*

class Transformation extends Model  
{  
 protected $fillable = [  
 'name', 'code'  
 ];  
 *// Связь с искусственным интеллектом через промежуточную таблицу ai\_transformations* public function artificialIntelligences()  
 {  
 return $this->belongsToMany(ArtificialIntelligence::class, 'ai\_transformations', 'transformation\_id', 'ai\_id');  
 }  
}

**User**

Модель User является одной из центральных, представляет пользователя системы (см. Листинг 28). Позволяет организовать связи один ко многим со статьями, обсуждениями и комментариями, где пользователь является автором. Также реализуется связь принадлежности к роли (belongsTo с моделью Role). Для безопасности поле password скрыто при сериализации, а пароль хранится в хэшированном виде.

*Листинг 28 – Модель User app/Http/Models/User.php*

class User extends *Authenticatable*{  
 use HasApiTokens, Notifiable, HasFactory;  
 public $timestamps = false; *// Отключаем временные метки* protected $fillable = [  
 'login', 'password', 'email', 'birthday', 'avatar\_url', 'role\_id',  
 ];  
 protected $hidden = [  
 'password', 'api\_token',  
 ];  
 *// Списки полей для скрытия* protected function casts(): array{  
 return [  
 'password' => 'hashed'  
 ];  
 }  
 *// Связь с ролью* public function role()  
 {  
 return $this->belongsTo(Role::class);  
 }  
 *// Связь с обсуждениями (как автор)* public function discussions()  
 {  
 return $this->hasMany(Discussion::class, 'author\_id');  
 }  
 *// Связь со статьями (как автор)* public function articles()  
 {  
 return $this->hasMany(Article::class, 'author\_id');  
 }  
 *// Связь с комментариями (как автор)* public function comments()  
 {  
 return $this->hasMany(Comment::class, 'user\_id');  
 }  
}

# **ГЛАВА 5. РАЗРАБОТКА Веб части**

## Шаблон

Был выбран встроенный шаблонизатор Blade в Laravel, он из себя представляет мощный и удобный инструмент для создания динамических представлений, который позволяет разделять общую структуру страниц и переиспользовать её в разных частях приложения.

В проекте реализован базовый шаблон /resources/views/layouts/layout.blade.php, являющийся основой для всех страниц веб-приложения. Шаблон состоит из трёх ключевых секций: header, main и footer.

В разделе <head> осуществляется подключение внешних ресурсов: иконка сайта через asset(), CSS-стилей и JS-скрипта. Заголовок страницы также формируется динамически с помощью директивы @yield('title'), что даёт возможность каждой странице иметь собственный заголовок.

В секции header реализовано навигационное меню, содержащее ссылки на главную страницу, раздел обсуждений, информацию об искусственном интеллекте и личный кабинет пользователя.

Секция main предназначена для отображения уникального контента каждой страницы. Через неё происходит вставка содержимого конкретной страницы в общий шаблон.

В нижней части страницы — в блоке footer — размещена информация о проекте, логотип и контактная почта.

Также в шаблоне реализована система уведомлений (toast-уведомления), которая отображает сообщения об успешных действиях или ошибках.

Итоговый интерфейс шаблона представлен на Рисунке X. Он включает в себя верхнее меню с навигацией, область для основного контента и информационный футер. Все внешние ресурсы (стили, скрипты и изображения) подключаются корректно, обеспечивая корректное отображение и функциональность страниц. Код шаблона представлен в Листинге 29.

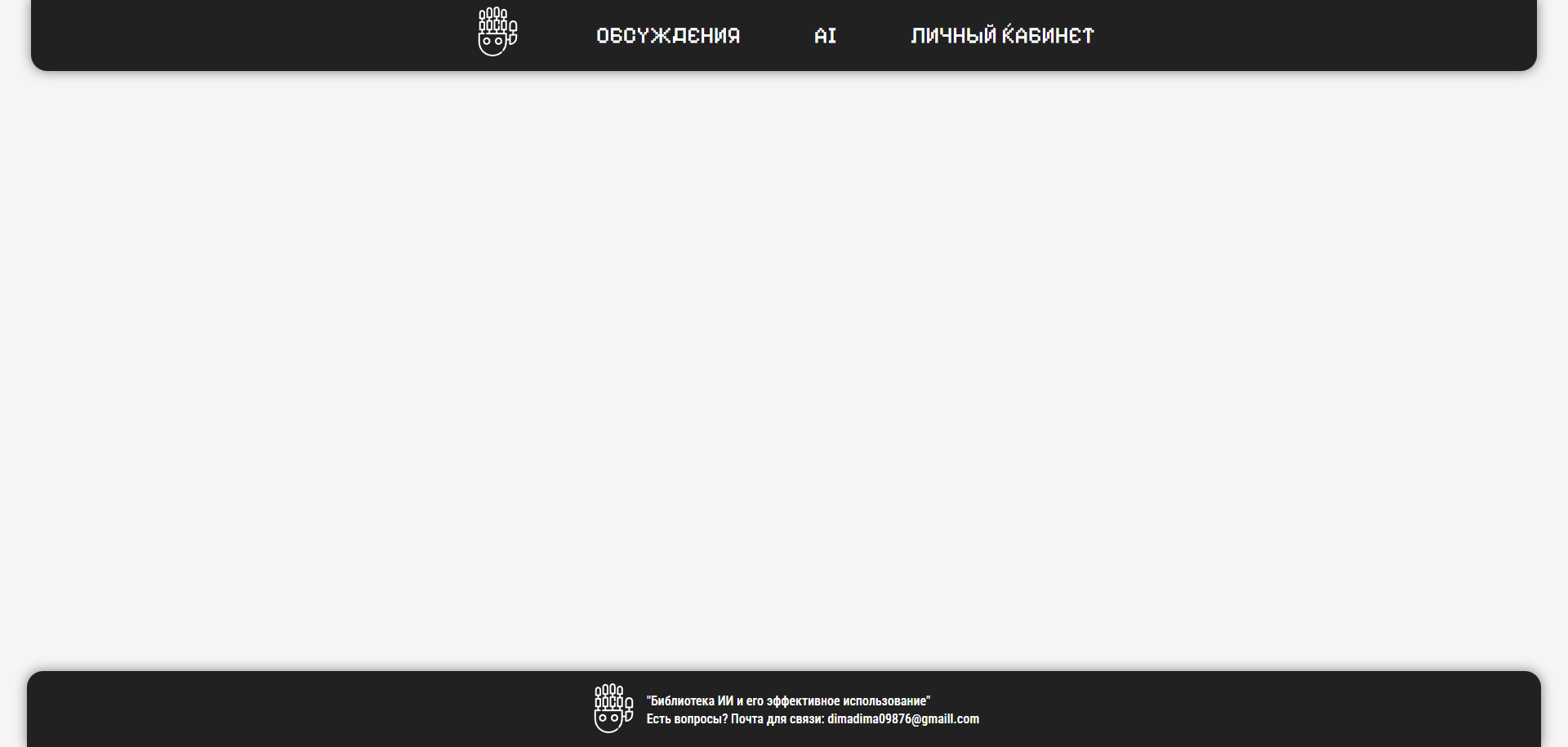


Рисунок Х – Шаблон

*Листинг 30 – Шаблон layout app/resources/views/layouts/layout.blade.php*

<!doctype html>  
<html lang="ru">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">  
 <meta http-equiv="X-UA-Compatible" content="ie=edge">  
 <link rel="icon" href="{{ asset('assets/images/logo.png') }}">  
 <link rel="stylesheet" href="{{ asset('assets/css/css.css') }}">  
 <script src="{{ asset('assets/js/main.js') }}"></script>  
 <title>@yield('title')</title>  
</head>  
<body>  
<div class="wrapper">  
 <header>  
 <nav>  
 <a href="{{ route('home') }}"><svg fill="#ffffff" height="61px" viewBox="0 0 512 " xml:space="preserve"…</a>  
 <a href="{{ route('discussions.index') }}">0бсуждения</a>  
 <a href="{{ route('ai.index') }}">AI</a>  
 <a href="{{ route('profile') }}">Личный кабинет</a>  
 </nav>  
 </header>  
 <main>  
 @yield('content')  
 </main>  
 <footer>  
 <div class="footer">  
 <div class="footer-1">  
 <a href="{{ route('home') }}"><svg fill="#ffffff" height="61px" viewBox="0 0 512 512" xml:space="preserve"…</svg></a>  
 </div>  
 <div class="footer-2">  
 <p>"Библиотека ИИ и его эффективное использование"</p>  
 <p>Есть вопросы? Почта для связи: dimadima09876@gmaill.com</p>  
 </div>  
 </div>  
 </footer>  
</div>  
@if(session('success') || session('error') || session('warning') || session('info'))  
 <div id="toast" class="toast @if(!session('success')) hidden @endif">  
 <div class="toast-content">  
 <span id="toast-message">  
 {{ session('success') ?: session('error') ?: session('warning') ?: session('info') }}  
 </span>  
 </div>  
 </div>  
@endif  
</body>  
</html>

# **ЗАКЛЮЧЕНИЕ**

# **СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

***Используемые интернет-источники:***

* Официальная документация Laravel / Режим доступа: <https://laravel.su/docs/11.x>, открытый.
* Руководство по PHP / Режим доступа: <https://metanit.com/php/tutorial/>, открытый.