МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

Специальность 1-40 05 01 03 «Информационные системы и технологии»

Специализация «Издательско-полиграфический комплекс»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ**

по дисциплине «Программирование в Internet»

Тема: Web-приложение «МедТестЧек»

Исполнитель

|  |  |  |
| --- | --- | --- |
| cтудент 4 курса группы 1  Руководитель |  | \_\_\_\_\_\_\_\_\_\_\_\_ Д. C. Шкабров  подпись, дата |
| ассистент |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. С. Кантарович  подпись, дата |
| Курсовой проект защищен с оценкой |  | \_\_\_\_\_\_\_\_\_\_\_\_ |
| Руководитель \_\_\_\_\_\_\_\_\_\_ |  | В. С. Кантарович |

подпись

Минск 2023

Содержание

[**Введение 5**](#_Toc154483542)

[**1 Постановка задач и обзор аналогов 6**](#_Toc154483543)

[**1.1 Обзор аналогов 6**](#_Toc154483545)

[**1.1.1 Веб-приложение «Kahoot!» 6**](#_Toc154483546)

[**1.1.2 Веб-приложение «Quizizz» 8**](#_Toc154483547)

[**1.2 Постановка задач 8**](#_Toc154483544)

[**1.3 Вывод по разделу 10**](#_Toc154483548)

[**2 Проектирование программного средства 11**](#_Toc154483549)

[**2.1 Выбор инструментов разработки 11**](#_Toc154483553)

[**2.2 Разработка модели базы данных 11**](#_Toc154483550)

[**2.3 Проектирование UML-диаграммы 15**](#_Toc154483551)

[**2.4 Проектирование алгоритмов 18**](#_Toc154483552)

[**2.5 Вывод……………………………………………………………………...20**](#_Toc154483554)

[**3 Разработка программного средства 21**](#_Toc154483555)

[**3.1 Разработка серверной части 21**](#_Toc154483556)

[**3.2 Разработка клиентской части 32**](#_Toc154483557)

[**3.3 Вывод……………………………………………………………………...35**](#_Toc154483558)

[**4 Тестирование 36**](#_Toc154483559)

[**4.1 Ручное тестирование 36**](#_Toc154483560)

[**4.2 Вывод……………………………………………………………………...38**](#_Toc154483561)

[**5 Руководство пользователя 39**](#_Toc154483562)

[**5.1 Руководство администратора 39**](#_Toc154483563)

[**5.2 Руководство преподавателя 40**](#_Toc154483564)

[**5.3 Руководство преподавателя 40**](#_Toc154483565)

[**5.4 Вывод……………………………………………………………………...41**](#_Toc154483566)

[**Заключение 42**](#_Toc154483567)

[**Список используемых источников 43**](#_Toc154483568)

[**ПРИЛОЖЕНИЕ А 44**](#_Toc154483569)

[**ПРИЛОЖЕНИЕ Б 49**](#_Toc154483570)

[**ПРИЛОЖЕНИЕ В 54**](#_Toc154483571)

[**ПРИЛОЖЕНИЕ Г 56**](#_Toc154483572)

[**ПРИЛОЖЕНИЕ Д 59**](#_Toc154483573)

[**ПРИЛОЖЕНИЕ Е 62**](#_Toc154483574)

[**ПРИЛОЖЕНИЕ Ж 65**](#_Toc154483575)

# **Введение**

В современной медицине постоянно появляются новые открытия и возможности, поэтому очень важно идти в ногу со временем и для этого надо постоянно обновлять свои знания. Тесты помогают восполнить пробелы и повысить уровень знаний, в соответсвии с современенностью.

Схемы автоматизации помогают управлять системой тестов и карточек, тем самым проверять знания у студентов-медиков, действующих врачей и даже тех, у кого был большой перерыв в работе.

Поэтому для составляния тестов, важно учесть базовые знания и новые тенденции, появляющиеся в медицине. Так пользователи приложения вспомнят то, что учили и узнают о своих пробелах.

Задачи таких тестов заинтересовать людей восполнять пробелы в своих знаниях и стремиться узнавать новое.

# **1 Постановка задач и обзор аналогов**

# **1.1 Обзор аналогов**

### **1.1.1 Веб-приложение «Kahoot!»**

Kahoot! [1] — игровая обучающая платформа, используемая в школах и других учебных заведениях. На сайте предоставлен каталог игр — «Kahoots» — каждая из которых является викториной, содержащей вопросы с несколькими вариантами ответов. Kahoot! был основан в 2012 сообща студентами и преподавателями Норвежского университета естественных и технических наук.

Kahoot! был разработан для групповых занятий. Учащиеся собираются вокруг общего экрана — например, интерактивной доски, проектора, монитора или экрана, передаваемого виртуально, например, через VNC, Jitsi, Skype или Google Hangouts. Игровой процесс простой: все игроки одновременно отвечают на вопросы на своих устройствах. Вопросы выводятся ученикам на экран по одному. Участники набирают очки за каждый правильный ответ. В конце викторины на экран выводится количество очков всех участников, набранных во время ответа на её вопросы.

Для участия в игре не требуется обязательной регистрации. Викторины можно выбрать из каталога на сайте. Также можно создавать новые викторины — часто эту задачу выполняет ведущий урок, например, учитель. Время ответа на каждый вопрос ограничено примерно 30-60 секундами.

Главная страница сайта «Kahoot!» представлена на рисунке 1.1.

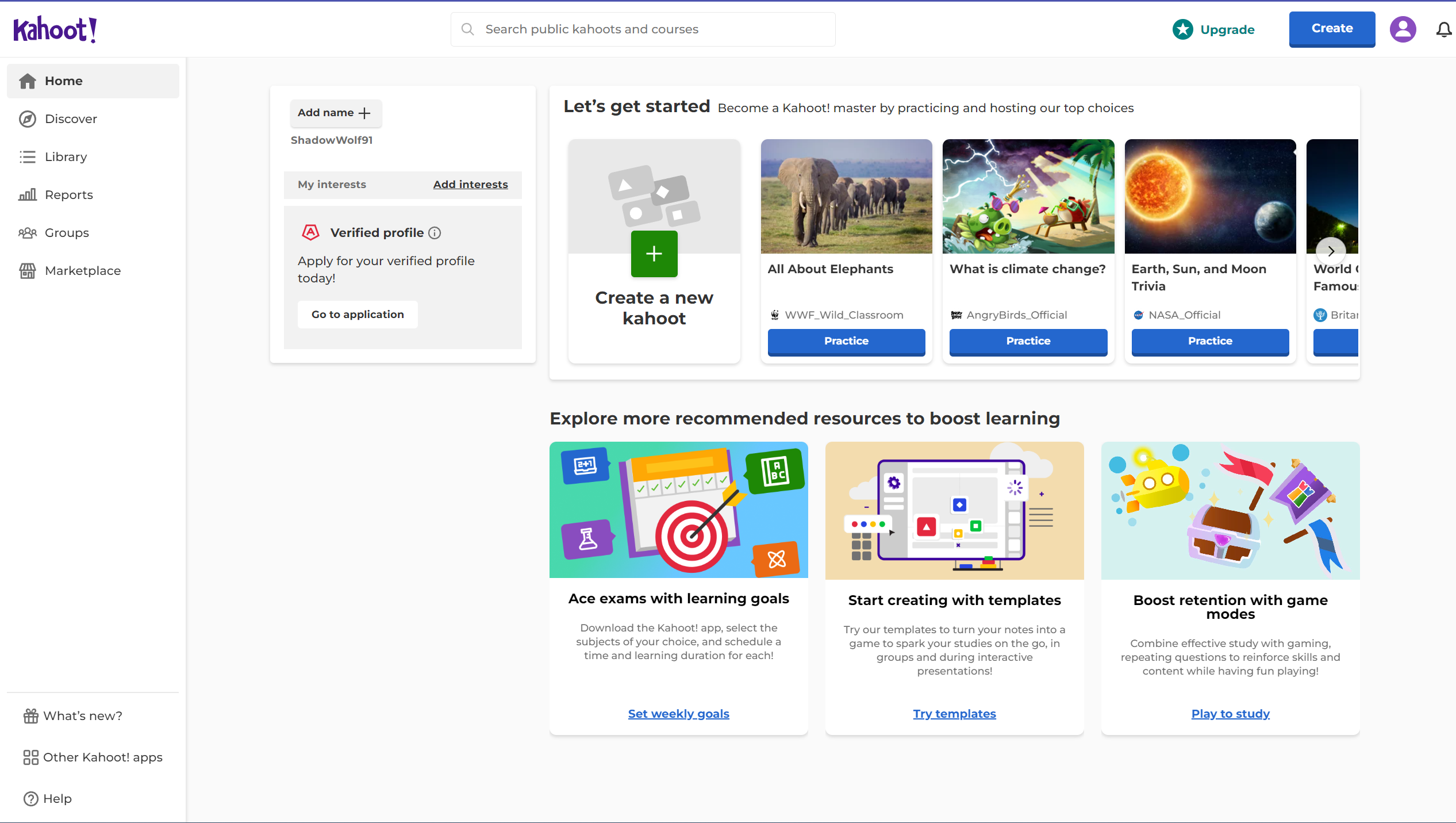


Рисунок 1.1 – Интерфейс «Kahoot!»

Страница с видом теста «Kahoot!» представлена на рисунке 1.2.

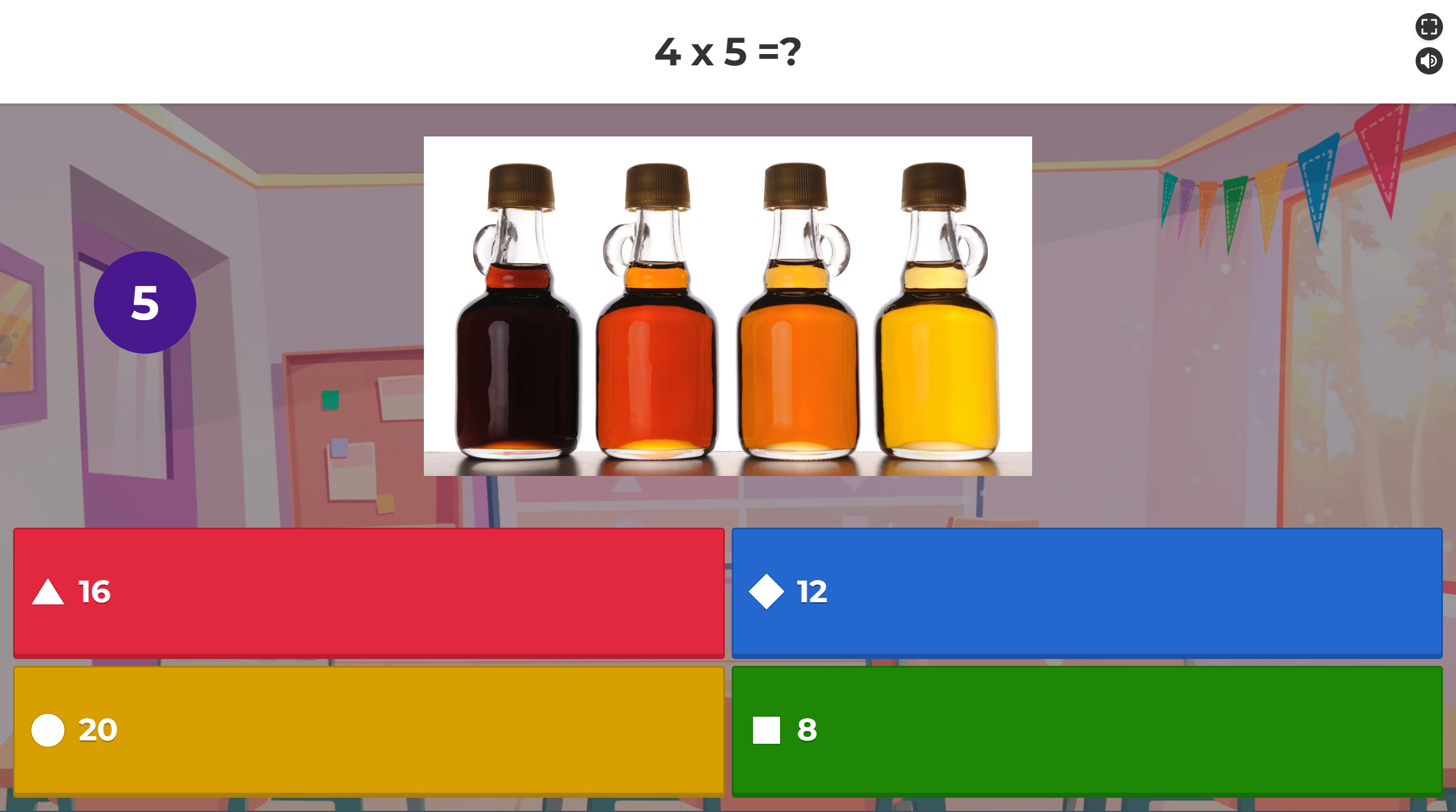


Рисунок 1.2 – Решение теста в «Kahoot!»

### **1.1.2 Веб-приложение «Quizizz»**

Quizizz [2]— это игровая платформа для взаимодействия со студентами, предлагающая множество функций, которые сделают занятия в классе веселыми, интерактивными и увлекательными.

Как учитель, вы можете проводить формативное оценивание, задавать домашние задания и осуществлять другие увлекательные взаимодействия со своими учениками (для всех классов).

Возможности викторины:

* Уроки/викторины, проводимые инструктором: Учителя контролируют темп; весь класс вместе просматривает каждый вопрос.
* Уроки/тесты, ориентированные на учащихся: учащиеся продвигаются в своем темпе, и вы видите таблицу лидеров и живые результаты по каждому вопросу или уроку.
* Доступ к миллионам викторин: импортируйте любые общедоступные викторины как есть, редактируйте и настраивайте их.
* (Викторина + урок) Редактор: выберите один из 5 различных типов вопросов, чтобы добавлять изображения, видео и аудио к телепортируемым вопросам из других викторин и уроков.
* Отчеты: получайте подробную информацию на уровне класса и учащихся по каждому тесту.

Главная страница сайта «Quizizz» представлена на рисунке 1.3.

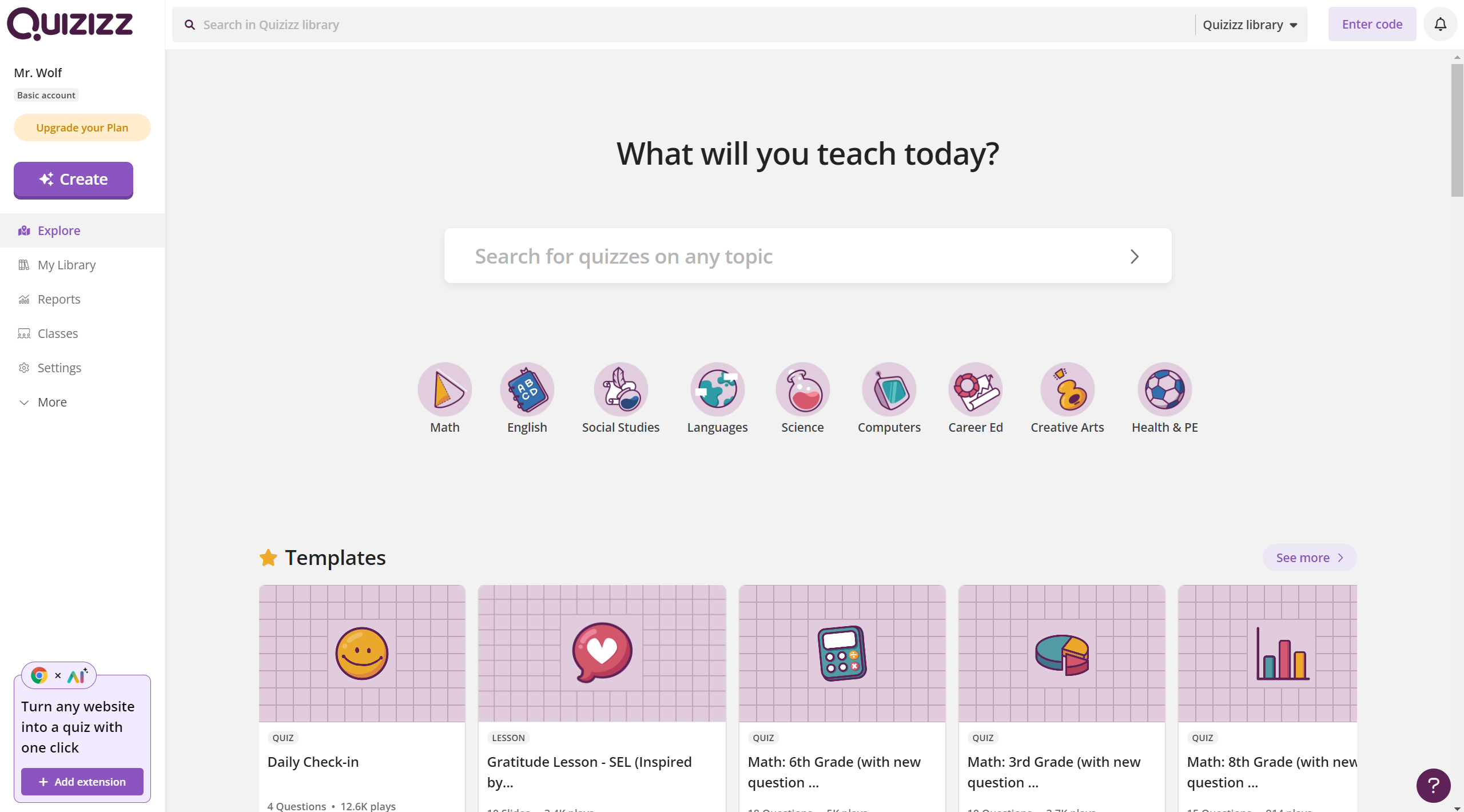


Рисунок 1.3 – Главная страница «Quizizz»

Страница сайта перед выполнением теста «Quizizz» представлена на рисунке 1.4.

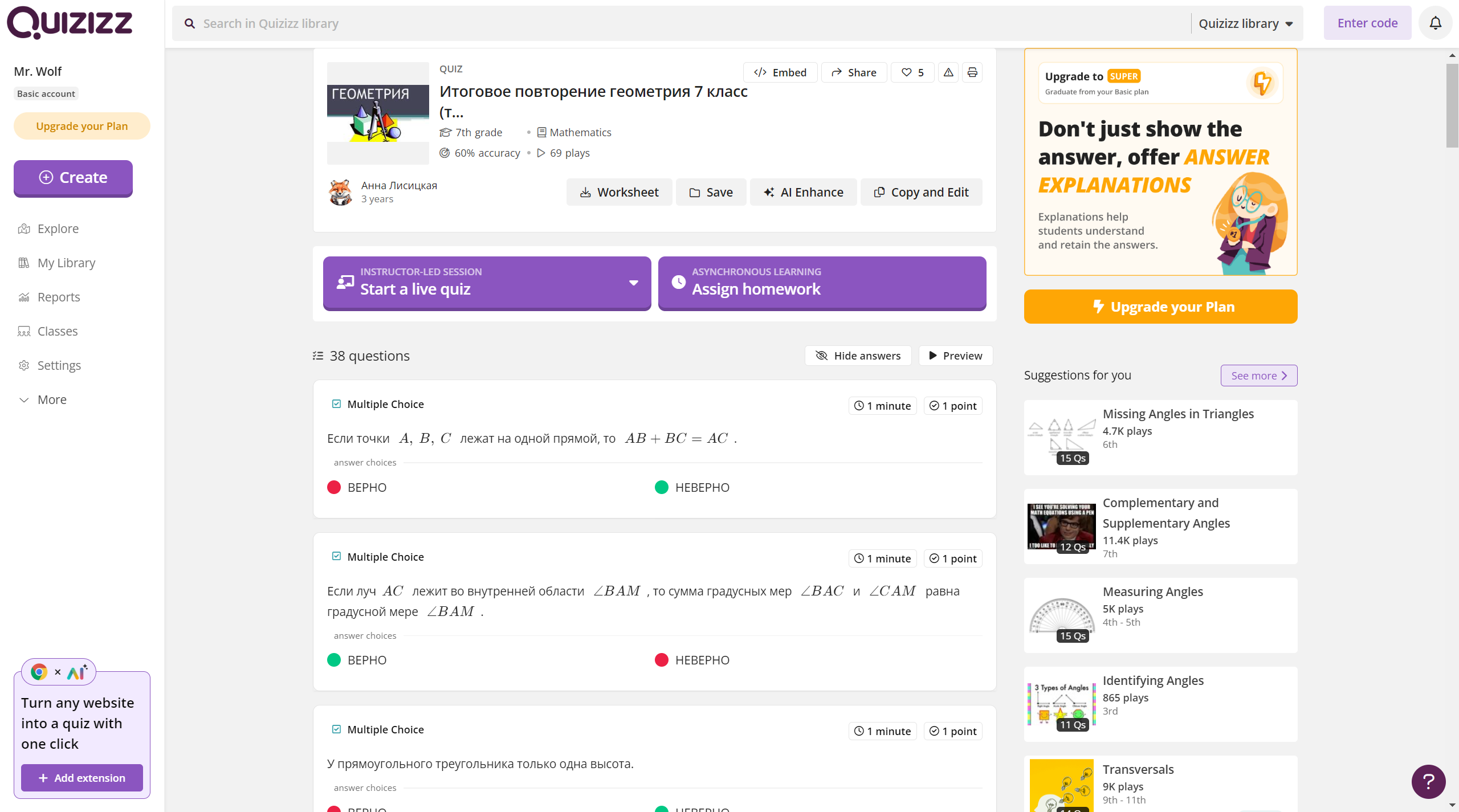


Рисунок 1.4 – Страница сайта с ответами перед выполнением теста «Quizizz»

## **1.2 Постановка задач**

Целью данного курсового проекта является создание веб-приложения для проверки знаний в виде тестов и других вариантов. Для реализации проекта необходимо установить список задач, выполнение которых будет отображать прогресс:

* обеспечивать реализацию 3 ролей: администратора, преподавателя и студента;
* обеспечивать регистрацию и авторизацию пользователей;
* обеспечивать возможность администратору управлять пользователями и предметами;
* обеспечивать преподавателю управление тестами: добавление, удаление, редактирование тестов;
* обеспечивать преподавателю и студенту результаты тестов;
* обеспечивать студенту выбор предмета, а затем выбор вида теста;
* обеспечивать студенту возможность создания или удаления теста по темам;
* обеспечивать возможность студенту видеть таймер во время ответов на вопросы.
  1. **Вывод по разделу**

В ходе анализа существующих аналогов, была выявлена целесообразность разработки и определен основной спектр функциональных возможностей, которые необходимо реализовать при разработке данного программного продукта. Также, на основании использования приложений-аналогов были выявлены основные критерии для построения дизайна приложения, позволяющего осуществлять наиболее простое и интуитивное его использование.

# **2 Проектирование программного средства**

После того как основной функционал был определен, можно начинать непосредственно проектирование приложения. Проектированию подлежит база данных приложения, а также UML-диаграмма и блок-схема алгоритмов работы.

**2.1 Выбор инструментов разработки**

Следующим логическим шагом следует выбор инструментов для разработки.

В качестве языка программирования для создания веб-приложения и интерфейса будет использоваться язык разработки TypeScript, также программное средство должно взаимодействовать с базой данных.

В ходе работы, вся логика, которая предусмотрена тематикой приложения, должна быть реализована на средствах, поставляемых выбранной СУБД и её стандартом языка SQL. В качестве основных инструментов, была выбрана СУБД PostgreSQL среда разработки для неё Prisma. Разработка самого веб-приложения осуществлялась на платформе node.js посредством модели MRCS.

PostgreSQL — свободная объектно-реляционная система управления базами данных. Существует в реализациях для множества UNIX-подобных платформ. Функции являются блоками кода, исполняемыми на сервере, а не на клиенте БД.

ASP.NET Core — свободно-распространяемый кроссплатформенный фреймворк для создания веб-приложений на платформе .NET с открытым исходным кодом.

MRCS — cтруктура каталогов Модель-Маршруты-Контроллеры-Сервисы. Для модульной структуры кода логика должна быть разделена на эти каталоги и файлы.

* Модели (Models) — определение схемы модели.
* Маршруты (Routes) — маршруты API сопоставляются с контроллерами.
* Контроллеры (Controllers) - контроллеры обрабатывают всю логику проверки параметров запроса, запроса и отправки ответов с правильными кодами.
* Службы (Services) - службы содержат запросы к базе данных и возвращают объекты или выдают ошибки.

## **2.2 Разработка модели базы данных**

Как отмечалось в предыдущих разделах, одним из ключевых моментов при проектировании и создании базы данных является грамотный анализ предметной области приложения. Как следствие – составление такой модели данных, которая будет правильно отражать то, как с этими с данными в общем, и этой моделью, в частности, подразумевается взаимодействовать.

Результатом корректно проведённого анализа, проектирования, и разработки, является модель, способная предоставить функционально все необходимые возможности для пользователя. В частности, для приложения по организации обучающих курсов, можно выделить набор таких операций, как:

* возможность администратору управлять пользователями и предметами;
* возможность преподавателю управлять тестами: добавление, удаление, редактирование тестов;
* возможность студенту выбор предмета, а затем выбор вида теста;
* возможность студенту создания или удаления теста по темам;

Основой инфраструктуры базы данных является грамотно спроектированная модель, которая отображает связь пользовательских таблиц. Правильное и корректное взаимодействие их друг с другом как раз и заключается в схеме базы данных со связями, верно отображающими их положение.

В базе данных реализовано восемь таблиц: «*User*», «*UserToken*», «*Statistics*», «*Subjects*», «*Themes*», «*Tests*», «*OpenQuestions*», «*Cards*». Между таблицами также настроены связи, отражающие их взаимосвязь.

Таблица «*User*» предназначена для хранения данных после прохождения регистрации пользователем. Именно из этой таблицы берутся данные для разграничения ролей и в зависимости от этого, от возможностей, выполняемых в приложении для разных пользователей.

Таблица «*UserToken*» предназначена для хранения токенов.

Таблица «*Statistics*» предназначена для хранения данных о статистике пользователя (отметка, набранные баллы, количество правильно отвеченных)

Таблица «*Subjects*» предназначена для хранения информации об учебном предмете: название учебного предмета.

Таблица «*Themes*» предназначена для хранения информации о темах учебного предмета, как название темы, выбор режима прохождения

Таблица «*Tests*» предназначена для хранения данных о тесте и там содержиться такая информация как: вопрос, ответы (A, B, C, D), правильный ответ, какой теме они принадлежат.

Таблица «*OpenQuestions*» предназначена для хранения данных об открытом вопросе и там содержиться такая информация как: вопрос, ответ, правильный ответ, какой теме они принадлежат.

Таблица «*Cards*» предназначена для хранения данных о карточке и там содержиться такая информация как: вопрос, ответ, правильный ответ, какой теме они принадлежат.

Общая структура базы данных представлена на рисунке 2.1 ниже.

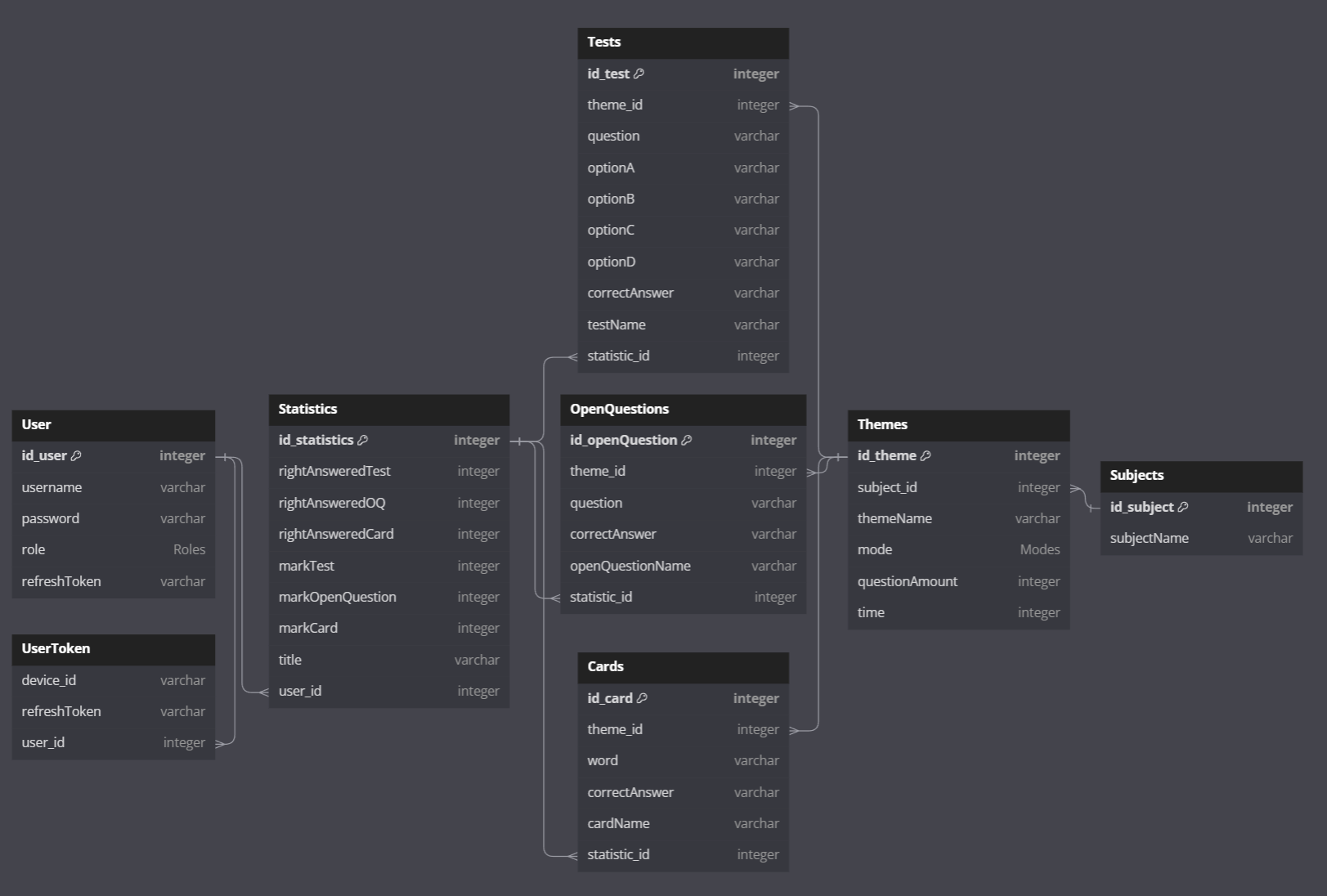


Рисунок 2.1 – Структура базы данных

В качестве примера, того, на чём основывались решения построения тех или иных связей, рассмотрим связи сущностей «*Tests*»/«*OpenQuestions*»/«*Cards*» и «*Statistics*», а также «*Tests*»/«*OpenQuestions*»/«*Cards*» и «*Themes*».

«*Themes*» и «*Tests*»/«*OpenQuestions*»/«*Cards*» являются центральными понятиями в проектируемой системе, так как в совокупности они представляют собой, в некотором смысле, информационное ядро системы, отражающее её основное предназначение – управление видами закрепления знаний и создания для них определенных типов. Связь между этими сущностями определена как «один ко многим». Это объясняется тем, что один тип курсов может содержать некоторое множество конкретных курсов, например, по разным направлениям математики.

Связь между сущностями «*Statistics*» и«*Tests*»/«*OpenQuestions*»/«*Cards*» отражает факт участия в решениях тестов/вопросов/карточек студентов – один студент может как проходить несколько тестов/вопросов/карточек. Соответственно, связь между ними должна быть отражена отношением «один ко многим».

Далее будут расписана каждая таблица для общего понимания, какие данные хранятся в них. Таблица «*Users*» содержит следующие поля:

- *id\_user* – идентификационный номер пользователя;

- *username* – логин пользователя для авторизации в приложении;

- *password* – хешированный пароль пользователя;

- *role* – тип пользователя;

Таблица «*UserToken*» содержит следующие поля:

- *device\_id* – идентификационный номер устройства;

- *refreshToken* – токен пользователя;

- *user\_id* – идентификационный номер пользователя;

Таблица «*Statistics*» основная информация о статистиках пользователей:

- *id\_statistics* – идентификационный номер статистики пользователя;

- *rightAnsweredTest* – количество правильно отвеченных в тесте;

- *rightAnsweredOQ* – количество правильно отвеченных в вопросах;

- *rightAnsweredCard* – количество правильно отвеченных в карточке;

- *markTest* – оценка за ответы в тесте;

- *markOpenQuestion* – оценка за ответы в вопросах;

- *markCard* – оценка за ответы в карточке;

- *title* - заголовок;

- *user\_id* - идентификационный номер пользователя;

Таблица «*Subjects*» содержит в себе конкретные курсы:

- *id\_subject* – идентификационный номер предмета;

- *subjectName* – название предмета;

Таблица «*Themes*» включает в себя информацию об учителях:

- *id\_theme* – идентификационный номер темы;

- *subject\_id* – идентификационный номер предмета;

- *themeName* – название темы;

- *mode* – режим проверки знаний;

- *questionAmount* – количество вопросов.

Таблица «*Tests*» нужна для получения информации о предметах:

- *id\_test* – идентификационный номер теста;

- *theme\_id* – идентификационный номер темы;

- *question* – вопрос;

- *optionA* – ответ A;

- *optionB* – ответ B;

- *optionC* – ответ C;

- *optionD* – ответ D;

- *correctAnswer* – правильный ответ;

- *testName* – название теста;

- *statistic\_id* – идентификационный номер статистики пользователя.

Таблица «*OpenQuestions*» содержит информацию о руководителях:

- *id\_openQuestion* – идентификационный номер руководителя;

- *theme\_id* – идентификационный номер темы;

- *question* – вопрос;

- *correctAnswer* - правильный ответ;

- *openQuestionName* - название открытого вопроса;

- *statistic\_id* - идентификационный номер статистики пользователя.

Таблицы «*Cards*» содержит с себе данные о студентах:

- *id\_card* – идентификационный номер студента;

- *theme\_id* – идентификационный номер темы;

- *word* - слово;

- *correctAnswer* - правильный ответ;

- *cardName* – название карточки;

- *statistic\_id* - идентификационный номер статистики пользователя.

Остальные примеры связей между сущностями являются довольно тривиальными в сравнении с вышеуказанными, и логика их проектирования не должна вызывать вопросов, исходя только из семантического отражения любой из сущностей.

На рисунке 2.1 проиллюстрированы следующие связи:

* связь «*User-Statistics*»: один ко многим;
* связь «*User-UserToken*»: один ко многим;
* связь «*Statistics-Tests*»: один ко многим;
* связь «*Statistics-OpenQuestions*»: один ко многим;
* связь «*Statistics-Cards*»: один ко многим;
* связь «*Subjects-Themes*»: один ко многим.
* связь «*Themes-Tests*»: один ко многим;
* связь «*Themes-OpenQuestions*»: один ко многим;
* связь «*Themes-Cards*»: один ко многим.

## **2.3 Проектирование UML-диаграммы**

При разработке программного обеспечения производится определение схем взаимодействия для представления, какие воздействия на систему смогут оказывать разные пользователи. Схемы взаимодействия — это описание возможного поведения пользователя и определение того, как система будет реагировать на его поведение и приспосабливаться к нему. И для этого используются UML-диаграммы.

Основная задача приложения — представлять собой единое средство, дающее возможность конечному пользователю использовать всю функциональность приложения и отслеживать поведение системы.

Данный этап является одним из самых значимых по время проектирования, так как именно тут решается, кто и какими ролями будет обладать в данной системе.

В данном приложении существует три роли: администратор, преподователь и студент. Но перед тем, как быть в роли, пользователям нужно пройти регистрацию и в данном случаем диаграмма будет иметь вид, как показано на рисунке 2.2.

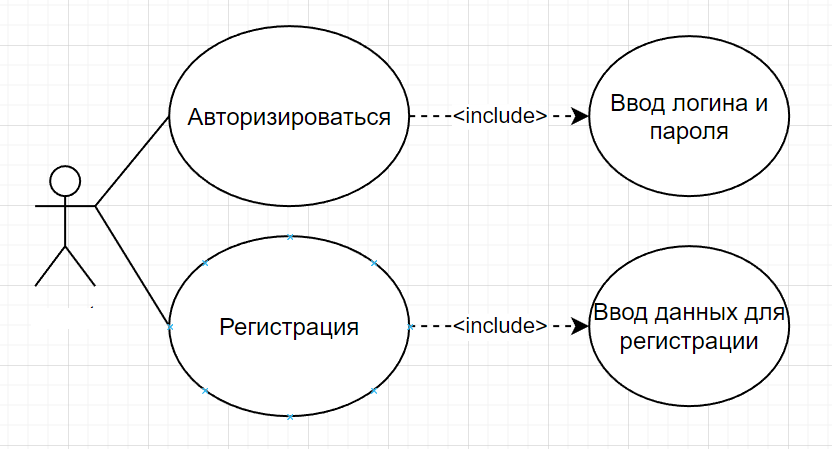


Рисунок 2.2 – UML-диаграмма регистрации и авторизации

В данной диаграмме показывается, что пользователь, который не вошел в систему должен пройти регистрацию и авторизацию и после этого будет доступен функционал приложения.

После регистрации или авторизации пользователь переходит в роль либо студента, либо преподователя, либо администратора, которым доступны функционалы, как показано на рисунке 2.3



Рисунок 2.3 – UML-диаграмма всех пользователей

Для начала рассмотрим пользователя студента. Он может выполнять выбор уже из готового контента и с ним взатмодействовать.

Как видно из диаграммы преподователь может выполнять множество функций в приложении по созданию разных видов проверки знаний, так же ему доступно и редактирование данных, и их удаление.

Если пользователь в системе авторизируются под особыми правами, детали чего мы опустим, то пользователь будет в роли администратора, которому доступно еще больше функционала.

Как видно, администратор наделен наибольшими правами по сравнению с другими пользователями системы, он может создавать все предметы с темами приложения, изменять их и в случае, если необходимо, то и удалять.

Каждая роль имеет своих участников, которые должны мочь выполнять только то, что предусмотрено для них системой, любое нарушение пределов данных и оговоренных в этом приложении ролей может привести к непредвиденным последствиям, что означает важность правильной реализации данный ролей. Реализация, таким образом, должна предусматривать, что каждый пользователь может совершать определенные действия, согласно его роли. Если пользователь пытается сделать что-то, что находиться вне его полномочий, система должна пресечь его действия, иначе они пагубно отразятся на системе.

## **2.4 Проектирование алгоритмов**

Важным этапом при взаимодействии с системой является регистрация пользователя, так как далее происходит процесс авторизации, который позволит использовать весь функционал приложения. Поэтому, на мой взгляд, важным этапом будет являться проверка правильности ввода данных пользователя при регистрации, таких как проверка на неповторимость логина, проверка правильно ли пользователь ввел пароль с первого раза. Блок-схема данного алгоритма изображена на рисунке 2.5.

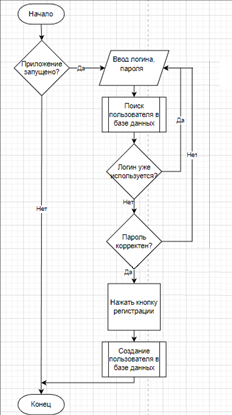


Рисунок 2.5 – Блок-схема алгоритма регистрации пользователя

Данный алгоритм гарантирует, что в приложении не будет двух пользователей с одинаковыми именами и что при повторном вводе пароля, пользователь не будет сомневаться, что он правильно ввел данные, которые будут сохранены в базе данных.

## **2.5 Вывод**

Таким образом в конце данной главы была спроектирована база данных, описаны UML-диаграммы вариантов использования для разных типов пользователей, как для администратора, так и для преподователя и студента, был описан алгоритм регистрации пользователей с хорошей проверкой на корректность введенных данных, а также определен стек технологий, используемый в данном проекте.

Таким образом, заранее известен список всех используемых технологий, сферы деятельности, которые они покрывают и некоторые основные алгоритмы, вокруг и на основе которых потом будут появляться новые механизмы взаимодействия между частями системы.

1. **Разработка программного средства**

Приложение будет разрабатываться посредством языка TypeScrtpt используя для реализации кроссплатформенную технологию node.js и модель MRCS.

MRCS — шаблон проектирования, с помощью которого его компоненты (модель данных приложения, маршруты приложения, службы приложения и взаимодействие с пользователем) разделены на четыре отдельных так, что модификация одного из них оказывает минимальное воздействие на остальные.

## **3.1 Разработка серверной части**

Архитектура программного обеспечения — совокупность важнейших решений об организации программной системы. Архитектура включает: выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов; соединение выбранных элементов структуры и поведения, во всё более крупные системы; архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

Структура проекта состоит из папок «*dist*», «*node\_modules*», «*prisma*», «*src*», а также файлов «*.env*», «*package.json*», «*.gitignore*», «*package-lock.json*» и «*tsconfig.json*»,.

Структура проекта изображена на рисунке 3.1.

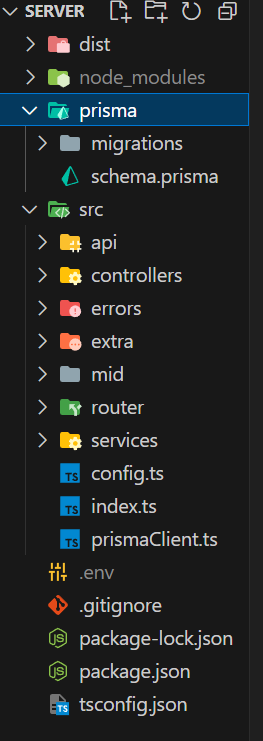


Рисунок 3.1 – Структура проекта

Папка «*prisma*» предназначена для моделирования данных. Каждая из этих моделей описывает таблицу в соответствующей базе данных и служит основой для сгенерированного доступа к данным с API, предоставляемого Prisma Client. В Schema.prisma описывается модель, а в папку migrations мигрирует эта модель в sql.

Папка «*src*» содержит в себе сам код сервера, где описана его логика и структура. В папке содержатся другие папки, как: «*api*», «*controllers», «errors*», «*extra*», «*mid*», *«router»* и *«service»,* а также файлов «*config.ts*», «*index.ts*» и «*.prismaClient.ts*».

В папке *«api»* описывается интерфейсы для сервера. А именно запросы и ответы такие как: создание, изменение, удаление, получение данных.

На рисунке 3.2 показано распределение интерфеймов, а таперь рассмотрим в пример интерфейс для сождания пользователя:

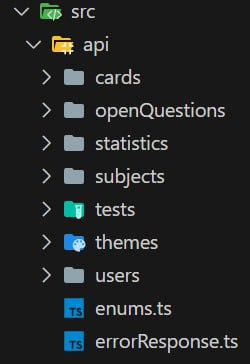


Рисунок 3.2 – Структура папки api

import { Roles } from "../../enums";

export interface ICreateUserRequest {

username: string;

password: string;

refreshToken?: string;

role?: keyof typeof Roles;

}

export interface ICreateUserResponse {

id\_user: number;

username: string;

password: string;

role: keyof typeof Roles;

refreshToken: string;

}

Листинг 3.3 – Структура интерфейса для создания пользователя

Папка *«сontroller»* отвечает за обработку системных событий, и при этом он не относится к интерфейсу пользователя. Controller определяет методы для выполнения системных операций.

Здесь уже происходит обработка всех событий и определяет методы, как create, update, delete, get (-Card).

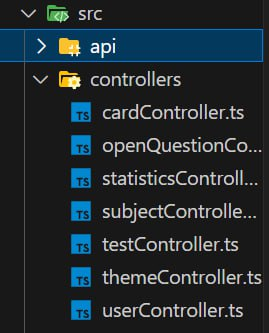


Рисунок 3.4 – Структура папки controller

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateCardRequest,

ICreateCardResponse,

} from "../api/cards/reg/createCard";

import {

IDeleteCardRequest,

IDeleteCardResponse,

} from "../api/cards/reg/deleteCard";

import {

IUpdateCardRequest,

IUpdateCardResponse,

} from "../api/cards/reg/updateCard";

import {

IGetAllCardsRequest,

IGetAllCardsResponse,

} from "../api/cards/reg/getAllCards";

import {

IGetCardByIdRequest,

IGetCardByIdResponse,

} from "../api/cards/reg/getCardById";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import CardService from "../services/cardService";

import UserRequestError from "../errors/userRequestError";

export default class CardController {

//get

static getAllCards: RequestHandler<

undefined,

IGetAllCardsResponse | IErrorResponse,

undefined,

IGetAllCardsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.getAllCards(req.query);

res.json({

cardsData: result,

cursor: result[result.length - 1]?.id\_card || null,

});

} catch (e) {

return next(e);

}

};

static getCardById: RequestHandler<

undefined,

IGetCardByIdResponse | IErrorResponse,

undefined,

IGetCardByIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.getCardById(req.query);

if (!result)

return next(

UserRequestError.NotFound(

CARD WITH ID ${req.query.id\_card} NOT FOUND

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

//create

static createCard: RequestHandler<

undefined,

ICreateCardResponse | IErrorResponse,

ICreateCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.createCard(req.body);

res.status(201).json(result);

} catch (e) {

return next(e);

}

};

//update

static updateCardData: RequestHandler<

undefined,

IUpdateCardResponse | IErrorResponse,

IUpdateCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.updateCardData(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

//delete

static deleteCard: RequestHandler<

undefined,

IDeleteCardResponse | IErrorResponse,

IDeleteCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await CardService.deleteCard(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}

Листинг 3.5 – Структура cardController.ts для карточки

В папке *«errors»* описывается класс и конструкторы для обработки ошибок на стороне клиента.

Папка *«extra»* испольуется для дополнительного написания кода, в нем описываюся фалы с функциями.

В папке *«mid»* написаныпромежуточные и связующее фрагменты кода в конвейере приложения, используемый для обработки запросов и ответов.

Папка *«router»* в себе содержит компоненты для с отслеживаниия состояния, который заставляет работать все остальные компоненты навигации и хуки, решение для переключения и маршрутизации страниц.

В router мы определим маршруты для модульной структуры нашего кода обработки маршрута и выполнения в контроллере методов и событий.

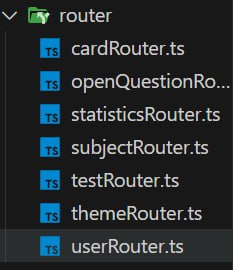


Рисунок 3.6 – Структура папки router

import { Router } from 'express'

import ThemeEndpoints from '../api/themes/endpoints'

import ThemeController from '../controllers/themeController'

const themeRouter = Router()

themeRouter.get(

ThemeEndpoints.GET\_BY\_ID,

ThemeController.getThemeById

)

themeRouter.get(

ThemeEndpoints.GET\_ALL\_THEMES,

ThemeController.getAllThemes

)

themeRouter.post(

ThemeEndpoints.CREATE,

ThemeController.createTheme

)

themeRouter.patch(

ThemeEndpoints.UPDATE,

ThemeController.updateTheme

)

themeRouter.delete(

ThemeEndpoints.DELETE,

ThemeController.deleteTheme

)

export default themeRouter

Листинг 3.7 – Структура themeRouter.ts для темы

И последняя папочка *«service»*, где инкапсулируется определенная бизнес-логика или функциональность приложения. Он предоставляет интерфейс для взаимодействия с другими компонентами приложения и абстрагирует детали реализации функциональности. Классы *service* обычно используются в объектно-ориентированном программировании для достижения разделения задач и для того, чтобы сделать код более модульным и удобным в обслуживании.

А здесь уже отдельная бизнес-логика для функциональности с предметом, (создание, удаление, обнавление, получение всех предметов и т.д.)

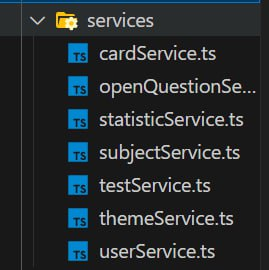


Рисунок 3.8 – Структура папки service

import { ICreateSubjectRequest } from "../api/subjects/reg/createSubject";

import { IDeleteSubjectRequest } from "../api/subjects/reg/deleteSubject";

import { IUpdateSubjectRequest } from "../api/subjects/reg/updateSubject";

import { IGetAllSubjectsRequest } from "../api/subjects/reg/getAllSubjects";

import { IGetSubjectBySubIdRequest } from "../api/subjects/reg/getSubjectBySubId";

import UserRequestError from "../errors/userRequestError";

import prismaClient from "../prismaClient";

export default class SubjectService {

//get

static getSubjectBySubId = async ({

id\_subject,

}: IGetSubjectBySubIdRequest) =>

prismaClient.subjects.findUnique({

where: { id\_subject: +id\_subject },

});

static getAllSubjects = async ({

cursor,

subjectName,

skip,

take,

}: IGetAllSubjectsRequest) =>

prismaClient.subjects.findMany({

skip,

take,

cursor: cursor ? { id\_subject: cursor } : undefined,

where: { subjectName: { contains: subjectName, mode: "insensitive" } },

});

//create

static createSubject = async ({ subjectName }: ICreateSubjectRequest) => {

const subject = await prismaClient.subjects.findUnique({

where: { subjectName },

select: { id\_subject: true },

});

if (subject)

throw UserRequestError.NotFound(

SUBJECT WITH NAME ${subjectName} NOT CREATED

);

return prismaClient.subjects.create({

data: {

subjectName,

},

});

};

//update

static updateSubject = async ({

id\_subject,

subjectName,

}: IUpdateSubjectRequest) => {

const subject = await prismaClient.subjects.findUnique({

where: { id\_subject },

select: { id\_subject: true },

});

if (!subject)

throw UserRequestError.NotFound(

SUBJECT WITH ID ${id\_subject} NOT FOUND

);

return prismaClient.subjects.update({

where: { id\_subject },

data: {

subjectName,

},

});

};

//delete

static deleteSubject = async ({ id\_subject }: IDeleteSubjectRequest) => {

const subject = await prismaClient.subjects.findUnique({

where: { id\_subject: id\_subject },

select: { id\_subject: true },

});

if (!subject)

throw UserRequestError.NotFound(

SUBJECT WITH ID ${id\_subject} NOT FOUND

);

return prismaClient.subjects.delete({

where: { id\_subject: id\_subject },

});

};

}

Листинг 3.9 – Структура subjectService.ts для предмета обучения

В приложение также имеется разграничение по ролям для пользователей. Так администратор имеет больше доступа к ресурсам приложения и может иметь допуск к страницам «Themes», «Subjects» и «Users», в пример как показано на рисунке 3.10.

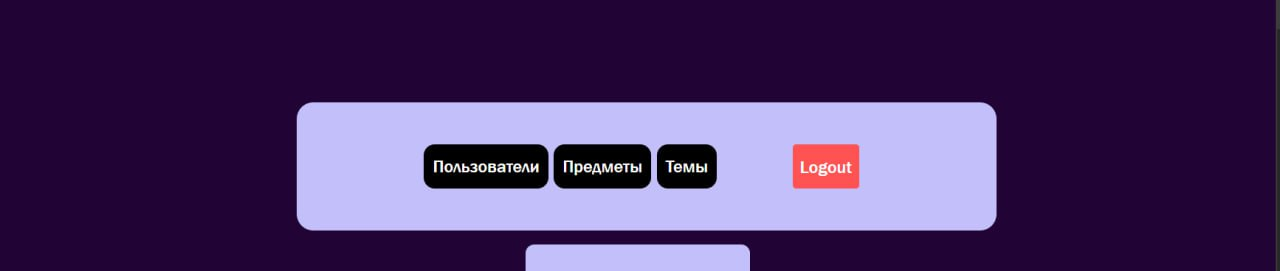


Рисунок 3.10 – Вкладки и страница доступная администратору

В свою же очередь, или посетитель сайта может видеть только часть из вкладок, доступных администратору и переходить только на них.

Данные действия в разнице доступа администратора, преподавателя и студента сделаны для того, чтобы пользователь не мог иметь доступ к данным, которые нежелательно для редактирования.

* 1. **Разработка клиентской части**

Клиентская часть приложения связана непосредственно с технологий TypeScript, например, с использованием React и валидацией данных.

Мы уже рассматривали сущности, но также здесь имеются атрибуты валидация для полей на страницах приложения, которые используются на клиентской стороне приложения. Пример атрибутов валидации можно увидеть в листинге 3.11.

return (

<div className={styles.modal}>

{selectedUser ? (

<>

<h3>Редактирование пользователя</h3>

<div className={styles.div}>

<p>Логин</p>

<input

type='text'

value={selectedUser.username}

className={`${styles.input} ${

!validData.username ? styles.error : ''

}`}

maxLength={30}

title={

'Логин должен содержать 4-30 латинских символов. ' +

'Можно использовать числа'

}

onChange={e => {

setValidData(prev => ({

...prev,

username: new RegExp(/^[a-zA-Z0-9]{4,30}$/).test(

e.target.value

),

}))

setSelectedUser(prev => ({

...prev,

username: e.target.value,

}))

}}

/>

</div>

<div className={styles.div}>

<p>Пароль</p>

<input

value={selectedUser.password}

type='password'

className={`${styles.input} ${

!validData.password ? styles.error : ''

}`}

maxLength={120}

title={

'Пароль должен содержать 8-120 латинских символов, включать символы ' +

'!@#$%^&\*, иметь Хотя бы одну заглавную букву'

}

onChange={e => {

setValidData(prev => ({

...prev,

password: new RegExp(

/(?=.\*[0-9])(?=.\*[!@#$%^&\*])(?=.\*[a-z])(?=.\*[A-Z])[0-9a-zA-Z!@#$%^&\*]{8,120}/g

).test(e.target.value),

}))

setSelectedUser(prev => ({

...prev,

password: e.target.value,

}))

}}

/>

</div>

<div>

<button

disabled={

!selectedUser

!validData.username

!validData.password ||

selectedUser.username === ''

}

onClick={async () => {

await updateUser(selectedUser)

setSelectedUser(null)

}}>

Сохранить

</button>

<button onClick={() => setSelectedUser(null)}>Отменить</button>

</div>

</>

Листинг 3.11 – Атрибуты валидации для создания пользователя

В данном случае у нас имеются атрибуты, которые указывают, что поля должны быть заполнены латинскими буквами и цифрами, размером 4 для имени пользавателя и для пароля в размере 8 символов, так же должны вводить по определенному синтаксису благодаря регулярным выражениям.

Данные атрибуты валидации будут наблюдаться практически у всех сущностей приложения, которые имеют аналогичный вид, чтобы пользователь корректно вводил свои данные.

Также для клиентской стороны приложения присуще задание кода HTML и TypeScript. Рассмотрим пример кода TypeScript и HTML, который предназначен для добавления и редактирования пользователей, который показан в листинге 3.17.

<div className={styles.div}>

<p>Название темы</p>

<input

type='text'

className={styles.input}

value={newTheme.themeName}

maxLength={20}

onChange={e =>

setNewTheme(prev => ({

...prev,

themeName: e.target.value,

}))

}

/>

</div>

<div className={styles.div}>

<p>Режим</p>

<select

name='unitsselect'

id='unitsselectnew'

value={newTheme.mode}

onChange={e =>

setNewTheme(prev => ({

...prev,

mode: e.target.value as Modes,

}))

}>

{Object.values(Modes).map(unit => (

<option value={unit}>{unit}</option>

))}

</select>

</div>

<div className={styles.div}>

<p>Кол-во вопросов</p>

<input

type='number'

step={1}

max={32767}

value={selectedTheme.questionAmount}

onChange={e =>

setNewTheme(prev => ({

...prev,

questionAmount: +e.target.value,

}))

}

/>

</div>

<div className={styles.buttons}>

<button

disabled={newTheme.themeName === ''}

onClick={async () => {

await createTheme(newTheme)

setNewTheme(newThemeInitState)

}}>

Сохранить

</button>

<button onClick={() => setNewTheme(newThemeInitState)}>

Отменить

</button>

</div>

</div>

)}

</div>

Листинг 3.12 – Код JavaScript для добавления и редактирования студента

## **3.3 Вывод**

По результату проведенной работы в данном разделе, было разработано приложение программно удовлетворяющее функционалу, определенному ранее. И в конце всего процесса был создан docker-контейнер для разворачивания программного средства.

После разработки стоит приступить к следующей не менее важной части разработки любого приложения – к тестированию. Так как, прежде чем оно увидит свет, оно должно быть протестировано в основных местах, чтобы при взаимодействии с пользователем были учтены и исправлены все ошибки.

1. **Тестирование**

Тестирование — направление, которое позволяет выявить подобные ошибки, проблемы и баги еще на пути массового тиражирования программы. Эксперт в мире тестирования ищет пробелы в коде, которые вызывают проблемы и заставляют программу работать не так, как задумал инженер-разработчик.

Основные задачи тестирования:

* выявить ошибки и оперативно их устранить;
* проверить соответствие продукта заявленным требованиям;
* оценить качество работы разработчиков;
* получить информацию, необходимую для принятия дальнейших решений.

Во время разработки было принято решено, провести ручное тестирование.

## **4.1 Ручное тестирование**

Тестирование включает в себе проверку правильности осуществления перехода между страницами приложения, которые совершают пользователи системы, также проверку правильности отправления запросов, их обработки, а также правильности отображение представлений.

Рассмотрим вариант авторизации пользователя, когда он вводит не верно, а также неправильно вводит пароль. При заполнении формы неправильными данными должно быть получено сообщение об ошибке. Заполнение формы неправильными данными изображено на рисунке 4.1.

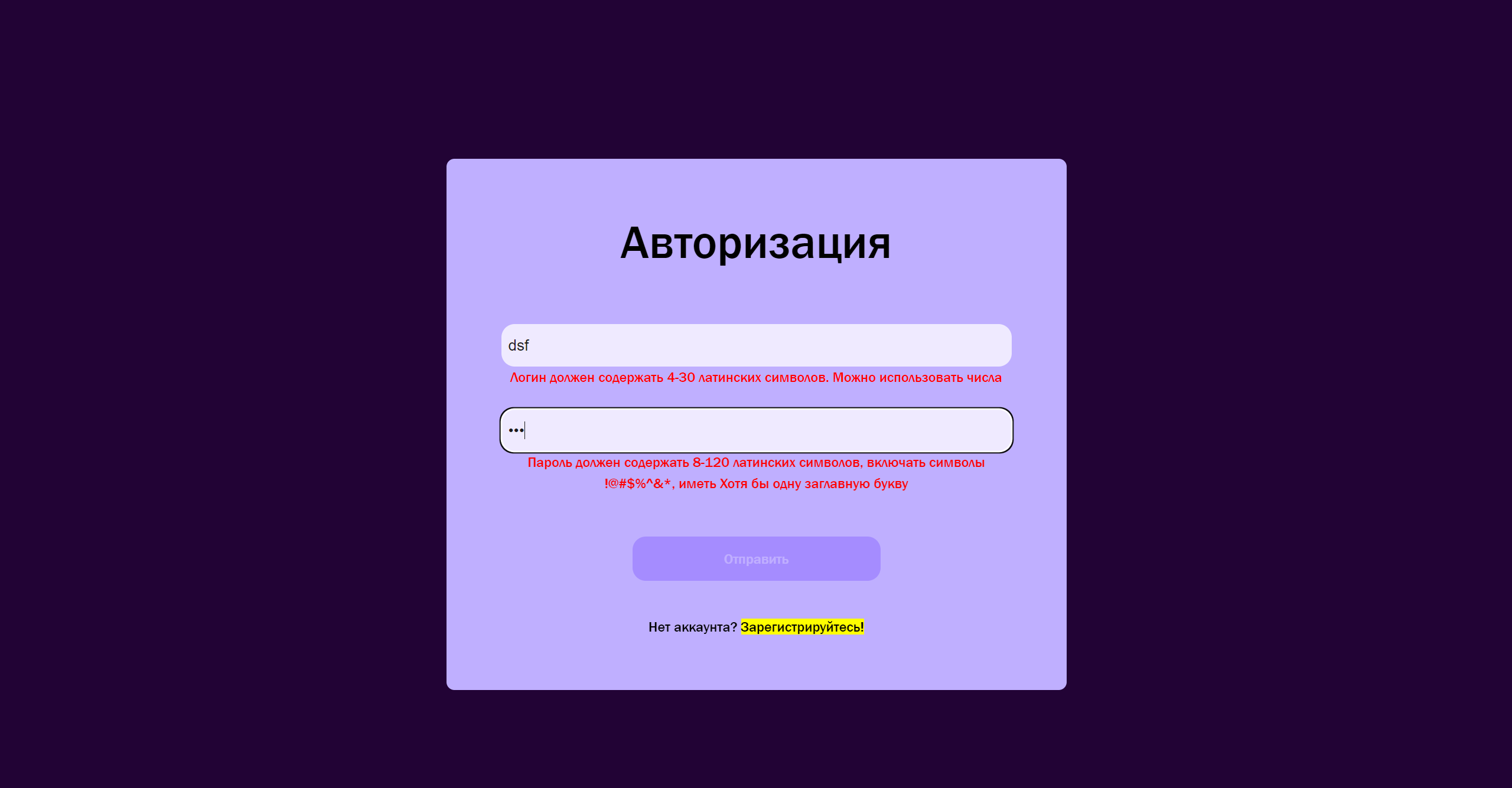


Рисунок 4.1 – Форма авторизации с неверными данными

Теперь заполним поля авторизации правильно, как показано на рисунке 4.2 для примера и войдем в систему.

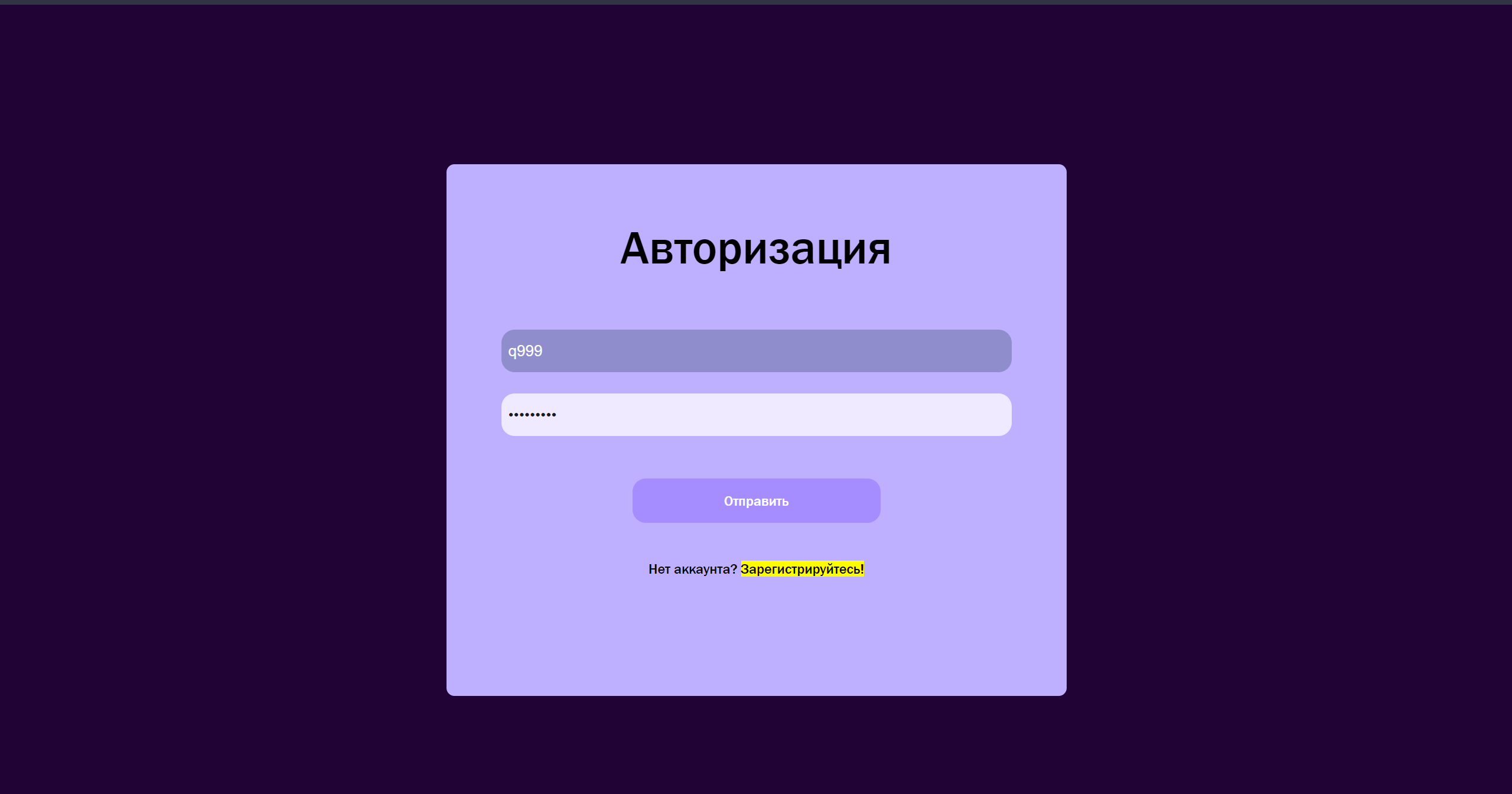


Рисунок 4.2 – Форма авторизации с верными данными

Далее проверим корректность ввода информации в поля приложения. Сделаем это на примере создания темы. Для создания темы нам нужно указать обязательно название темы. Нам нужно проверить корректность добавления темы, если поле название темы, не введено, то система не дает сохранить тему, как это показано на рисунке 4.3, а если введено, то можно нажать на кнопку, как это показано на рисунке 4.4.

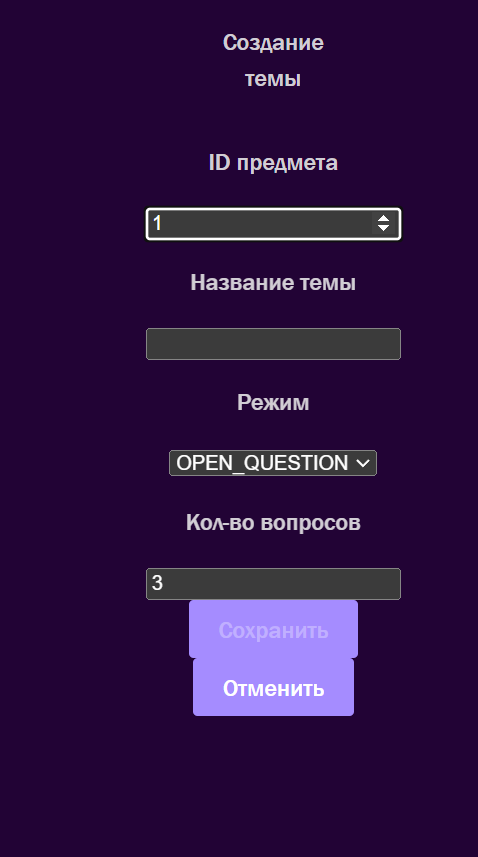
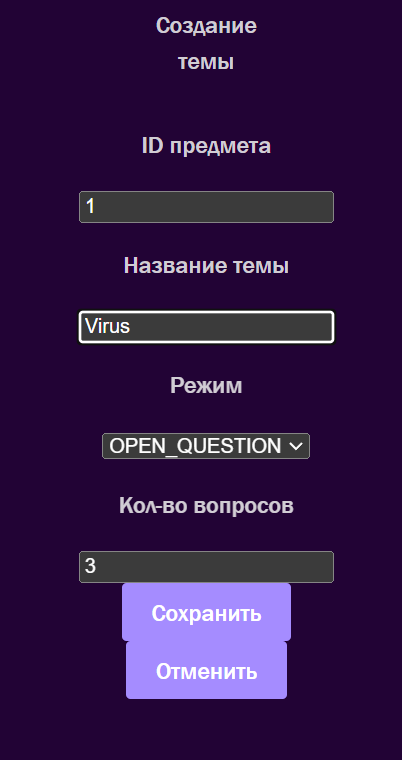
 

Рисунок 4.3 и 4.4 – Форма создания темы

## **4.2 Вывод**

После проведения ручного тестирования, можно сделать вывод, что покрываемый функционал работает правильно и никаких ошибок в ходе исполнения замечено не было. Все функции приложения отрабатывают корректно, а в случае, если возникает ошибка пользователь получает соответствующее уведомление. В свою очередь это говорит о грамотном проектировании приложения.

# **5 Руководство пользователя**

Главной задачей курсового проекта является создание веб-приложения, которое бы позволяло создать обучающие курсы. Для этого программно было реализовано с помощью Visual Studio взаимодействие с элементами страниц приложения. В данном руководстве описано как пользователю работать с веб-приложением, которое имеет первоначальный вид авторизации, представленный на рисунке 5.1.

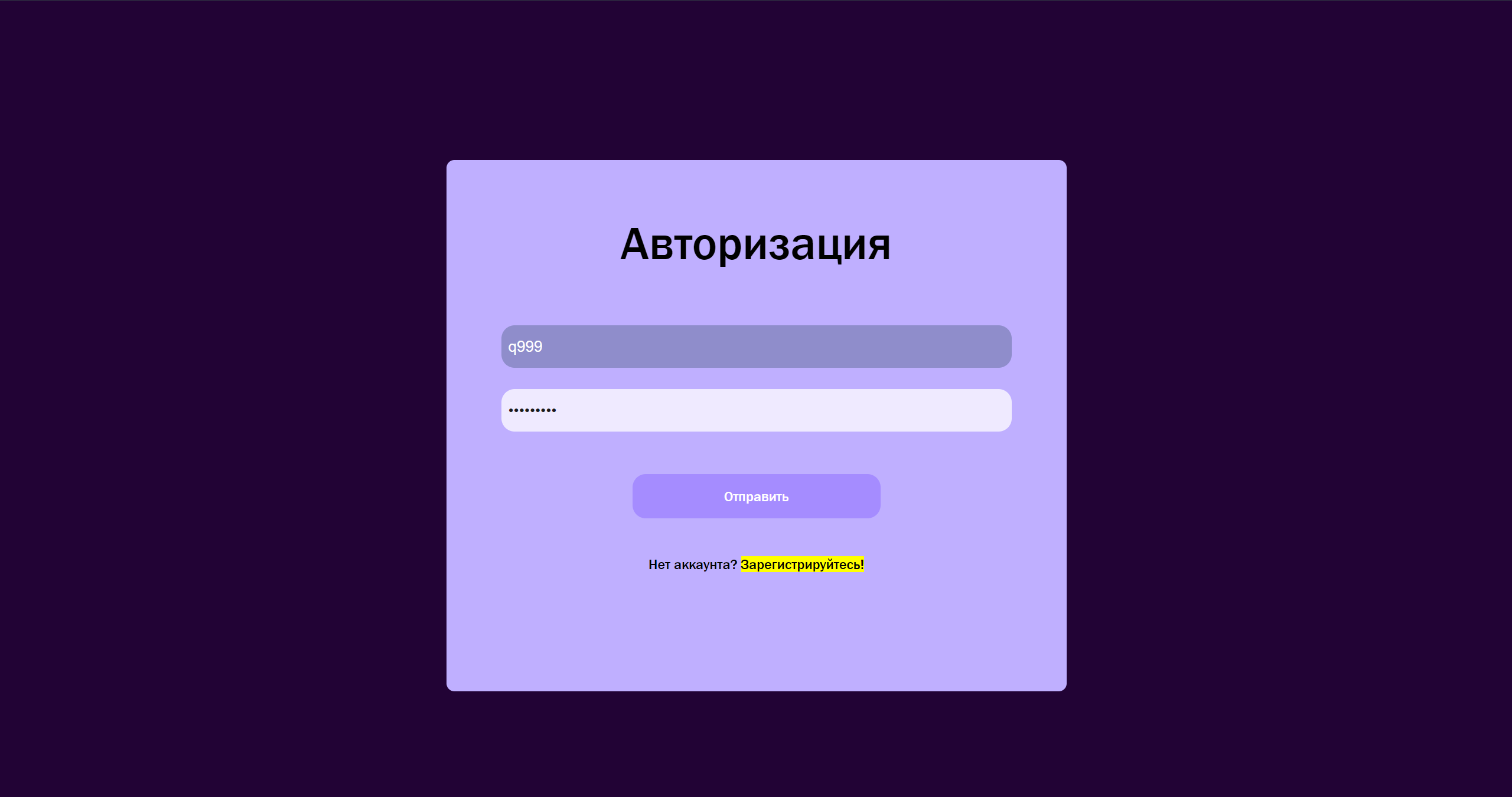


Рисунок 5.1 – Страница авторизации

Также новый пользователь может войди в систему зарегестривавшись в ней. Администратор приложения и преподаватель создается по умолчанию при запуске программы. Чтобы войти под этой ролью необходимо знать логин и пароль, которые соответственно будут admin, admin и teacher, teacher.

Далее рассмотрим руководство для пользователя и для администратора, которые будут отличаться в доступе к некоторым вкладкам приложения, а также к возможностям редактирования и удаления некоторых данных.

## **5.1 Руководство администратора**

В меню мы имеем несколько вкладок, показана на рисунке 5.3, пользователь может посмотреть созданные предметы, пользователи и темы, создать новые предметы пользователей и темы, когда нажмет кнопку «Сохранить». Можно также переключаться между вкладками, найти их и выйти с аккаунта. Так как администратор наделен привилегированными правами, то он может и удалять, и редактировать предметы пользователей и темы.

В меню мы имеем несколько вкладок, которые показаны на рисунке 5.3.

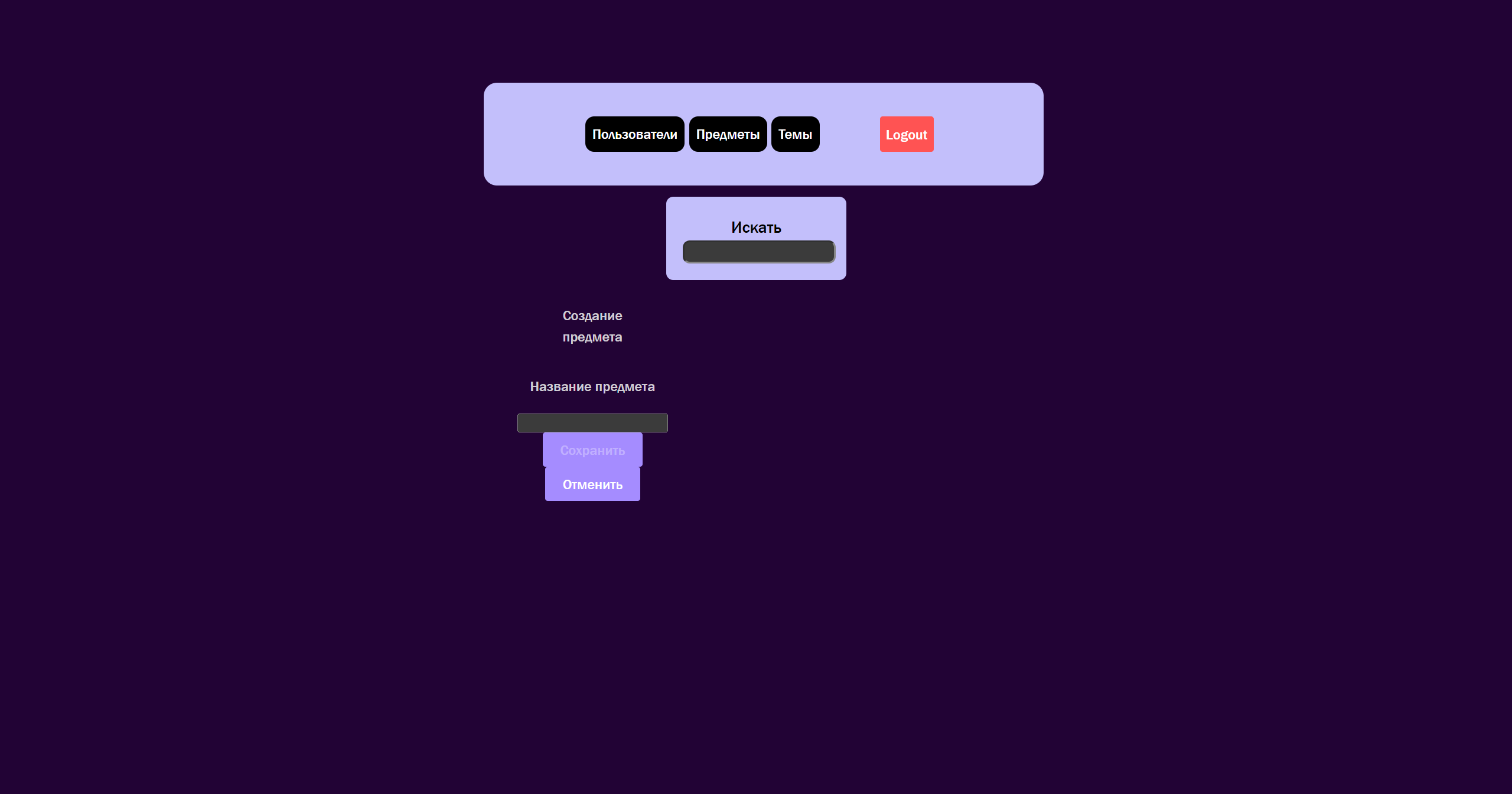


Рисунок 5.2 – Страница админа

## **5.2 Руководство преподавателя**

У преподавателя все то же самое, только у него другие вкладки, вместо пользавателей, предметов и темы, вкладки тесты, карточки и открытые вопросы.

## **5.3 Руководство преподавателя**

Аналогичным образом действует все у студента, только он не создает, не удаляет и не обнавленет, а только полученает, пользуется и выполняет все, что создал преподаватель.

Работоспособность веб-приложения было успешно проверено в браузере, например, Brave и Google Chrome.

## **5.4 Вывод**

Подводя итог руководства пользователя, можно отметить, что весь необходимый функционал для администрирования был предоставлен в полном объеме, как и функционал приложения. А также пользователь-учитель обладает широким доступом к ресурсам для работы с веб-приложением.

Заключение

В ходе выполнения курсового проекта мной были рассмотрены основные аналоги разрабатываемого программного средства. Были выявлены основные недостатки рассматриваемых систем. Затем был сформирован набор функциональных требований, покрывающих необходимый функционал.

Для реализации приложения использовалась платформа node.js. Node.js может работать поверх кроссплатформенной среды, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS, Linux.

Была разработана реляционная база данных для веб-приложения, использующая экземпляр PostgreSQL, имеющая в своем наборе 7 таблиц.

Были созданы дополнительные конфигурационные файлы, позволяющие быстро и легко администрировать, расширять функционал приложения.

В результате проведенной работы, был создан проект, который удовлетворяет потребностям разных пользователей и корректным образом выполняет все необходимые функции.

В данном курсовом мы также познакомились с Docker, подсистемой Windows для Linux, создавали Images и контейнеры.

Также я получила как опыт проектирования, так и опыт разработки приложения. Приобрела опыт работы с большим стеком технологий и изучила новые.

Опыт развертывания приложений, полученный во время выполнения курсового проекта, считаю немаловажным дополнением ко всем приобретённым навыкам разработки программного средства.

# **Список используемых источников**

1. Prisma: Полноценный ORM для Node.js и TypeScript [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/588562/>. – Дата доступа: 15.12.2023.

2. How To Use NodeJS Service Class [Электронный ресурс]. – Режим доступа: <https://medium.com/@soulaimaneyh/how-to-use-nodejs-service-class-a1e919a5cbd8>. – Дата доступа: 16.12.2023.

3. Node JS — Router and Routes [Электронный ресурс]. – Режим доступа: https://medium.com/swlh/node-js-router-and-routes-a4a3cfced5c4. – Дата доступа: 14.12.2023.

4. Quizizz [Электронный ресурс]. – Режим доступа: https://quizizz.com/?lng=en. – Дата доступа: 05.10.2023.

5. Kahoot! [Электронный ресурс]. – Режим доступа: https://kahoot.com/. – Дата доступа: 05.10.2023.

6. Руководство по Docker [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/ruvds/blog/450312//. – Дата доступа: 17.12.2023.

ПРИЛОЖЕНИЕ А

Листинг контроллера «cardController»

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateCardRequest,

ICreateCardResponse,

} from "../api/cards/reg/createCard";

import {

IDeleteCardRequest,

IDeleteCardResponse,

} from "../api/cards/reg/deleteCard";

import {

IUpdateCardRequest,

IUpdateCardResponse,

} from "../api/cards/reg/updateCard";

import {

IGetAllCardsRequest,

IGetAllCardsResponse,

} from "../api/cards/reg/getAllCards";

import {

IGetCardByIdRequest,

IGetCardByIdResponse,

} from "../api/cards/reg/getCardById";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import CardService from "../services/cardService";

import UserRequestError from "../errors/userRequestError";

export default class CardController {

//get

static getAllCards: RequestHandler<

undefined,

IGetAllCardsResponse | IErrorResponse,

undefined,

IGetAllCardsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.getAllCards(req.query);

res.json({

cardsData: result,

cursor: result[result.length - 1]?.id\_card || null,

});

} catch (e) {

return next(e);

}

};

static getCardById: RequestHandler<

undefined,

IGetCardByIdResponse | IErrorResponse,

undefined,

IGetCardByIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.getCardById(req.query);

if (!result)

return next(

UserRequestError.NotFound(

`CARD WITH ID ${req.query.id\_card} NOT FOUND`

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

//create

static createCard: RequestHandler<

undefined,

ICreateCardResponse | IErrorResponse,

ICreateCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.createCard(req.body);

res.status(201).json(result);

} catch (e) {

return next(e);

}

};

//update

static updateCardData: RequestHandler<

undefined,

IUpdateCardResponse | IErrorResponse,

IUpdateCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await CardService.updateCardData(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

//delete

static deleteCard: RequestHandler<

undefined,

IDeleteCardResponse | IErrorResponse,

IDeleteCardRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await CardService.deleteCard(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}

ПРИЛОЖЕНИЕ Б

Листинг контроллера «openQuestionController»

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateOpenQuestionRequest,

ICreateOpenQuestionResponse,

} from "../api/openQuestions/reg/createOpenQuestion";

import {

IDeleteOpenQuestionRequest,

IDeleteOpenQuestionResponse,

} from "../api/openQuestions/reg/deleteOpenQuestion";

import {

IUpdateOpenQuestionRequest,

IUpdateOpenQuestionResponse,

} from "../api/openQuestions/reg/updateOpenQuestion";

import {

IGetAllOpenQuestionsRequest,

IGetAllOpenQuestionsResponse,

} from "../api/openQuestions/reg/getAllOpenQuestions";

import {

IGetOpenQuestionByIdRequest,

IGetOpenQuestionByIdResponse,

} from "../api/openQuestions/reg/getOpenQuestionById";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import OpenQuestionsService from "../services/openQuestionService";

import UserRequestError from "../errors/userRequestError";

export default class OpenQuestionController {

//get

static getAllOpenQuestions: RequestHandler<

undefined,

IGetAllOpenQuestionsResponse | IErrorResponse,

undefined,

IGetAllOpenQuestionsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await OpenQuestionsService.getAllOpenQuestions(req.query);

res.json({

openQuestionsData: result,

cursor: result[result.length - 1]?.id\_openQuestion || null,

});

} catch (e) {

return next(e);

}

};

static getOpenQuestionById: RequestHandler<

undefined,

IGetOpenQuestionByIdResponse | IErrorResponse,

undefined,

IGetOpenQuestionByIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await OpenQuestionsService.getOpenQuestionById(req.query);

if (!result)

return next(

UserRequestError.NotFound(

`OPEN QUESTION WITH ID ${req.query.id\_openQuestion} NOT FOUND`

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

//create

static createOpenQuestion: RequestHandler<

undefined,

ICreateOpenQuestionResponse | IErrorResponse,

ICreateOpenQuestionRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await OpenQuestionsService.createOpenQuestion(req.body);

res.status(201).json(result);

} catch (e) {

return next(e);

}

};

//update

static updateOpenQuestionData: RequestHandler<

undefined,

IUpdateOpenQuestionResponse | IErrorResponse,

IUpdateOpenQuestionRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await OpenQuestionsService.updateOpenQuestionData(

req.body

);

res.json(result);

} catch (e) {

return next(e);

}

};

//delete

static deleteOpenQuestions: RequestHandler<

undefined,

IDeleteOpenQuestionResponse | IErrorResponse,

IDeleteOpenQuestionRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await OpenQuestionsService.deleteOpenQuestions(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}

# ПРИЛОЖЕНИЕ В

Листинг контроллера «statisticsController»

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

IUpdateStatisticRequest,

IUpdateStatisticResponse,

} from "../api/statistics/reg/updateStatistics";

import {

IGetAllStatisticsRequest,

IGetAllStatisticsResponse,

} from "../api/statistics/reg/getAllStatistics";

import {

IGetStatisticByUserIdRequest,

IGetStatisticByUserIdResponse,

} from "../api/statistics/reg/getStatisticByUserId";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import StatisticService from "../services/statisticService";

import UserRequestError from "../errors/userRequestError";

export default class StatisticsController {

//get

static getAllStatistics: RequestHandler<

undefined,

IGetAllStatisticsResponse | IErrorResponse,

undefined,

IGetAllStatisticsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await StatisticService.getAllStatistics(req.query);

res.json({

statisticsData: result,

cursor: result[result.length - 1]?.id\_statistics || null,

});

} catch (e) {

return next(e);

}

};

static getStatisticByUserId: RequestHandler<

undefined,

IGetStatisticByUserIdResponse[] | IErrorResponse,

IGetStatisticByUserIdRequest

// undefined,

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await StatisticService.getStatisticByUserId(req.body);

if (!result) return next(UserRequestError.NotFound(""));

res.json(result);

} catch (e) {

return next(e);

}

};

//update

static updateStatisticsData: RequestHandler<

undefined,

IUpdateStatisticResponse | IErrorResponse,

IUpdateStatisticRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await StatisticService.updateStatisticsData(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

}

# ПРИЛОЖЕНИЕ Г

Листинг контроллера «subjectController»

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateSubjectRequest,

ICreateSubjectResponse,

} from "../api/subjects/reg/createSubject";

import {

IDeleteSubjectRequest,

IDeleteSubjectResponse,

} from "../api/subjects/reg/deleteSubject";

import {

IGetSubjectBySubIdRequest,

IGetSubjectBySubIdResponse,

} from "../api/subjects/reg/getSubjectBySubId";

import {

IUpdateSubjectRequest,

IUpdateSubjectResponse,

} from "../api/subjects/reg/updateSubject";

import {

IGetAllSubjectsRequest,

IGetAllSubjectsResponse,

} from "../api/subjects/reg/getAllSubjects";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import SubjectService from "../services/subjectService";

import UserRequestError from "../errors/userRequestError";

export default class SubjectController {

static getSubjectBySubId: RequestHandler<

undefined,

IGetSubjectBySubIdResponse | IErrorResponse,

undefined,

IGetSubjectBySubIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await SubjectService.getSubjectBySubId(req.query);

if (!result)

return next(

UserRequestError.NotFound(

`SUBJECT WITH ID ${req.query.id\_subject} NOT FOUND`

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

static getAllSubjects: RequestHandler<

undefined,

IGetAllSubjectsResponse | IErrorResponse,

undefined,

IGetAllSubjectsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await SubjectService.getAllSubjects(req.query);

res.json({

subjectsData: result,

cursor: result[result.length - 1]?.id\_subject || null,

});

} catch (e) {

return next(e);

}

};

//create

static createSubject: RequestHandler<

undefined,

ICreateSubjectResponse | IErrorResponse,

ICreateSubjectRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await SubjectService.createSubject(req.body);

res.status(201).json({

...result,

});

} catch (e) {

return next(e);

}

};

//update

static updateSubjectData: RequestHandler<

undefined,

IUpdateSubjectResponse | IErrorResponse,

IUpdateSubjectRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await SubjectService.updateSubject(req.body);

res.json({

...result,

});

} catch (e) {

return next(e);

}

};

//delete

static deleteSubject: RequestHandler<

undefined,

IDeleteSubjectResponse | IErrorResponse,

IDeleteSubjectRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await SubjectService.deleteSubject(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}

# ПРИЛОЖЕНИЕ Д

Листинг контроллера «testController»

import { RequestHandler } from "express";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateTestRequest,

ICreateTestResponse,

} from "../api/tests/reg/createTest";

import {

IDeleteTestRequest,

IDeleteTestResponse,

} from "../api/tests/reg/deleteTest";

import {

IUpdateTestRequest,

IUpdateTestResponse,

} from "../api/tests/reg/updateTest";

import {

IGetAllTestsRequest,

IGetAllTestsResponse,

} from "../api/tests/reg/getAllTests";

import {

IGetTestByIdRequest,

IGetTestByIdResponse,

} from "../api/tests/reg/getTestById";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import TestsService from "../services/testService";

import UserRequestError from "../errors/userRequestError";

export default class TestController {

//get

static getAllTests: RequestHandler<

undefined,

IGetAllTestsResponse | IErrorResponse,

undefined,

IGetAllTestsRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await TestsService.getAllTests(req.query);

res.json({

testsData: result,

cursor: result[result.length - 1]?.id\_test || null,

});

} catch (e) {

return next(e);

}

};

static getTestById: RequestHandler<

undefined,

IGetTestByIdResponse | IErrorResponse,

undefined,

IGetTestByIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await TestsService.getTestById(req.query);

if (!result)

return next(

UserRequestError.NotFound(

`TEST WITH ID ${req.query.id\_test} NOT FOUND`

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

//create

static createTest: RequestHandler<

undefined,

ICreateTestResponse | IErrorResponse,

ICreateTestRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await TestsService.createTest(req.body);

res.status(201).json(result);

} catch (e) {

return next(e);

}

};

//update

static updateTestData: RequestHandler<

undefined,

IUpdateTestResponse | IErrorResponse,

IUpdateTestRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await TestsService.updateTestData(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

//delete

static deleteTest: RequestHandler<

undefined,

IDeleteTestResponse | IErrorResponse,

IDeleteTestRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await TestsService.deleteTest(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}

# ПРИЛОЖЕНИЕ Е

Листинг контроллера «themeController»

import { RequestHandler } from 'express'

import { IErrorResponse } from '../api/errorResponse'

import {

ICreateThemeRequest,

ICreateThemeResponse,

} from '../api/themes/reg/createTheme'

import {

IDeleteThemeRequest,

IDeleteThemeResponse,

} from '../api/themes/reg/deleteTheme'

import {

IGetThemeByIdRequest,

IGetThemeByIdResponse,

} from '../api/themes/reg/getThemeById'

import {

IUpdateThemeRequest,

IUpdateThemeResponse,

} from '../api/themes/reg/updateTheme'

import {

IGetAllThemesRequest,

IGetAllThemesResponse,

} from '../api/themes/reg/getAllThemes'

import callUnprocessableEntity from '../extra/callUnprocessableEntity'

import getValidationResult from '../extra/getValidationResult'

import ThemeService from '../services/themeService'

export default class ThemeController {

//get

static getThemeById: RequestHandler<

undefined,

IGetThemeByIdResponse | IErrorResponse,

undefined,

IGetThemeByIdRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req)

if (errorData) return callUnprocessableEntity(next, errorData)

try {

const result = await ThemeService.getThemeById(req.query)

res.json(result)

} catch (e) {

return next(e)

}

}

static getAllThemes: RequestHandler<

undefined,

IGetAllThemesResponse | IErrorResponse,

undefined,

IGetAllThemesRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req)

if (errorData) return callUnprocessableEntity(next, errorData)

try {

const result = await ThemeService.getAllThemes(req.query)

res.json({

themesData: result,

cursor: result[result.length - 1]?.id\_theme || null,

})

} catch (e) {

return next(e)

}

}

//create

static createTheme: RequestHandler<

undefined,

ICreateThemeResponse | IErrorResponse,

ICreateThemeRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req)

if (errorData) return callUnprocessableEntity(next, errorData)

try {

const result = await ThemeService.createTheme(req.body)

res.status(201).json(result)

} catch (e) {

return next(e)

}

}

//update

static updateTheme: RequestHandler<

undefined,

IUpdateThemeResponse | IErrorResponse,

IUpdateThemeRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req)

if (errorData) return callUnprocessableEntity(next, errorData)

try {

const result = await ThemeService.updateTheme(req.body)

res.json(result)

} catch (e) {

return next(e)

}

}

//delete

static deleteTheme: RequestHandler<

undefined,

IDeleteThemeResponse | IErrorResponse,

IDeleteThemeRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req)

if (errorData) return callUnprocessableEntity(next, errorData)

try {

await ThemeService.deleteTheme(req.body)

res.json({ count: 1 })

} catch (e) {

return next(e)

}

}

}

ПРИЛОЖЕНИЕ Ж

Листинг контроллера «userController»

import { RequestHandler } from "express";

import { createHash } from "node:crypto";

import { IErrorResponse } from "../api/errorResponse";

import {

ICreateUserRequest,

ICreateUserResponse,

} from "../api/users/reg/createUser";

import {

IDeleteUsersRequest,

IDeleteUsersResponse,

} from "../api/users/reg/deleteUsers";

import {

IDeleteUserRequest,

IDeleteUserResponse,

} from "../api/users/reg/deleteUser";

import {

IGetAllUsersRequest,

IGetAllUsersResponse,

} from "../api/users/reg/getAllUsers";

import {

IGetUserByUsernameRequest,

IGetUserByUsernameResponse,

} from "../api/users/reg/getUserByUsername";

import {

IGetUserTokensRequest,

IGetUserTokensResponse,

} from "../api/users/reg/getUserTokens";

import {

ILoginUserRequest,

ILoginUserResponse,

} from "../api/users/reg/loginUser";

import {

IUpdateUserDataRequest,

IUpdateUserDataResponse,

} from "../api/users/reg/updateUserData";

import UserRequestError from "../errors/userRequestError";

import callUnprocessableEntity from "../extra/callUnprocessableEntity";

import getValidationResult from "../extra/getValidationResult";

import Tokenizator from "../extra/tokenizator";

import UserService from "../services/userService";

export default class UserController {

//login

static loginUser: RequestHandler<

undefined,

ILoginUserResponse | IErrorResponse,

ILoginUserRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const user = await UserService.getUserByUsername(req.body);

if (!user)

return next(

UserRequestError.NotFound(`USER ${req.body.username} NOT FOUND`)

);

if (

createHash("sha512").update(req.body.password).digest("hex") !==

user.password

)

return next(UserRequestError.BadRequest("WRONG PASSWORD"));

const { refreshToken } = Tokenizator.generateTokens({

username: user.username,

role: user.role,

});

res

.cookie("refreshToken", refreshToken, {

maxAge: 30 \* 24 \* 60 \* 60 \* 1000,

httpOnly: true,

})

.json({

refreshToken,

username: user.username,

role: user.role,

id\_user: user.id\_user,

});

} catch (e) {

return next(e);

}

};

//get

static getUserByUsername: RequestHandler<

IGetUserByUsernameRequest,

IGetUserByUsernameResponse | IErrorResponse

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await UserService.getUserByUsername(req.params);

if (!result)

return next(

UserRequestError.NotFound(

`USER WITH LOGIN ${req.params.username} NOT FOUND`

)

);

res.json(result);

} catch (e) {

return next(e);

}

};

static getAllUsers: RequestHandler<

undefined,

IGetAllUsersResponse | IErrorResponse,

undefined,

IGetAllUsersRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await UserService.getAllUsers(req.query);

res.json({

usersData: result,

cursor: result[result.length - 1]?.id\_user || null,

});

} catch (e) {

return next(e);

}

};

static getUserTokens: RequestHandler<

undefined,

IGetUserTokensResponse[] | IErrorResponse,

undefined,

IGetUserTokensRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await UserService.getUserTokens(req.query);

res.json(result);

} catch (e) {

return next(e);

}

};

//create

static createUser: RequestHandler<

undefined,

ICreateUserResponse | IErrorResponse,

ICreateUserRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const { refreshToken } = Tokenizator.generateTokens(req.body);

const result = await UserService.createUser({

...req.body,

refreshToken,

});

res.cookie("refreshToken", refreshToken, {

maxAge: 30 \* 24 \* 60 \* 60 \* 1000,

httpOnly: true,

});

res.status(201).json(result);

} catch (e) {

return next(e);

}

};

//update

static updateUserData: RequestHandler<

undefined,

IUpdateUserDataResponse | IErrorResponse,

IUpdateUserDataRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await UserService.updateUserData(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

//delete

static deleteUsers: RequestHandler<

undefined,

IDeleteUsersResponse | IErrorResponse,

IDeleteUsersRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

const result = await UserService.deleteUsers(req.body);

res.json(result);

} catch (e) {

return next(e);

}

};

static deleteUser: RequestHandler<

undefined,

IDeleteUserResponse | IErrorResponse,

IDeleteUserRequest

> = async (req, res, next) => {

const errorData = getValidationResult(req);

if (errorData) return callUnprocessableEntity(next, errorData);

try {

await UserService.deleteUser(req.body);

res.json({ count: 1 });

} catch (e) {

return next(e);

}

};

}