Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ

УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

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

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

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**КУРСОВОЙ РАБОТЫ**

по дисциплине «Администрирование информационных систем и web-порталов».

Тема «Администрирование веб-приложения «МедТестЧек» и его развертывание при помощи Docker»

Исполнитель

студент 4 курса 1 группы Шкабров Д.С.

(подпись) (инициалы, фамилия)

Руководитель

ассистент Сазонова Д.В.

(подпись) (инициалы, фамилия)

Курсовая работа защищена с оценкой

Руководитель

ассистент Сазонова Д.В.

(подпись) (инициалы, фамилия)

Минск 2023

**Содержание**

[Введение 4](#_Toc122027028)

[1 Аналитический обзор технологий проекта 5](#_Toc122027029)

[1.1 Этап разработки веб-приложения 5](#_Toc122027030)

[1.2 Администрирование проекта 1](#_Toc122027031)8

[1.3 Начало работы 19](#_Toc122027032)

[2 Проверка работоспособности приложения 23](#_Toc122027033)

[Заключение 27](#_Toc122027034)

[Список используемых источников 28](#_Toc122027035)

**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.

Чтобы принять любое решение необходимо обладать необходимой для этого информацией, обычно количественной. Для этого создают хранилища данных. Хранилище данных (англ. Data Warehouse) — предметно-ориентированная информационная база данных, специально разработанная и предназначенная для подготовки отчётов и бизнес-анализа с целью поддержки принятия решений в организации. Хранилище данных строится на базе систем управления базами данных и систем поддержки принятия решений.

Принципы организации хранилища:

- Проблемно-предметная ориентация. Данные объединяются в категории и хранятся в соответствии с областями, которые они описывают, а не с приложениями, которые они используют;

- интегрированность. Данные объединены так, чтобы они удовлетворяли всем требованиям предприятия в целом, а не единственной функции бизнеса;

В идеале процесс пополнения представляет собой просто добавление новых данных без изменения прежней информации, уже находящейся в хранилище. Основной целью данной курсовой работы будет создание руководства администратора веб-сайта сервиса «МедТестЧек» на базе Docker. Для этого необходимо создать хранилище данных, в котором будут располагаться данные о предмтах, их темах и видов закрепления знаний на каждый из них.

Основными задачами будут являться:

- Документирование процесса установки веб-сайта сервиса «МедТестЧек» и необходимых для его работы вспомогательных компонентов;

- документирование доступных администратору на установленном веб-сайте возможностей на «МедТестЧек».

**1 Аналитический обзор технологий проекта**

При разработке проекта, первым делом стоит провести анализ аналогов, чтобы иметь представление о требуемых функциях и разделений по ролям в проекте.

После просмотра аналогов, их функций и работы в целом, можно вынести основные потребности в функционале:

* 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.

Также необходимо разработать приемлемый интерфейс, который подходит для пользователя любого уровня, а значит этот интерфейс должен быть простым и понятным для пользователей. Самой главной проблемой из рассматриваемых аналогов является реклама, а также отсутствие основополагающих данных, которые касаются непосредственно записей. Приложение должно работать безотказно и не позволять пользователю создавать некорректные записи, нарушающие общую структуру.

Также было принято решение добавить в приложение администратора, который сможет создавать, редактировать и если необходимо, то удалять пользователей и темы.

А еще есть роль преподователя, который и создает темы, тесты, карточки и вопросы для пользователей.

* 1. **Этап разработки веб-приложения**

Следующим логическим шагом следует выбор инструментов для разработки.

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.

Согласно схеме, были выделены следующие сущности:

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.

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 1.1 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.

Общая структура базы данных представлена на рисунке 1.1 ниже.

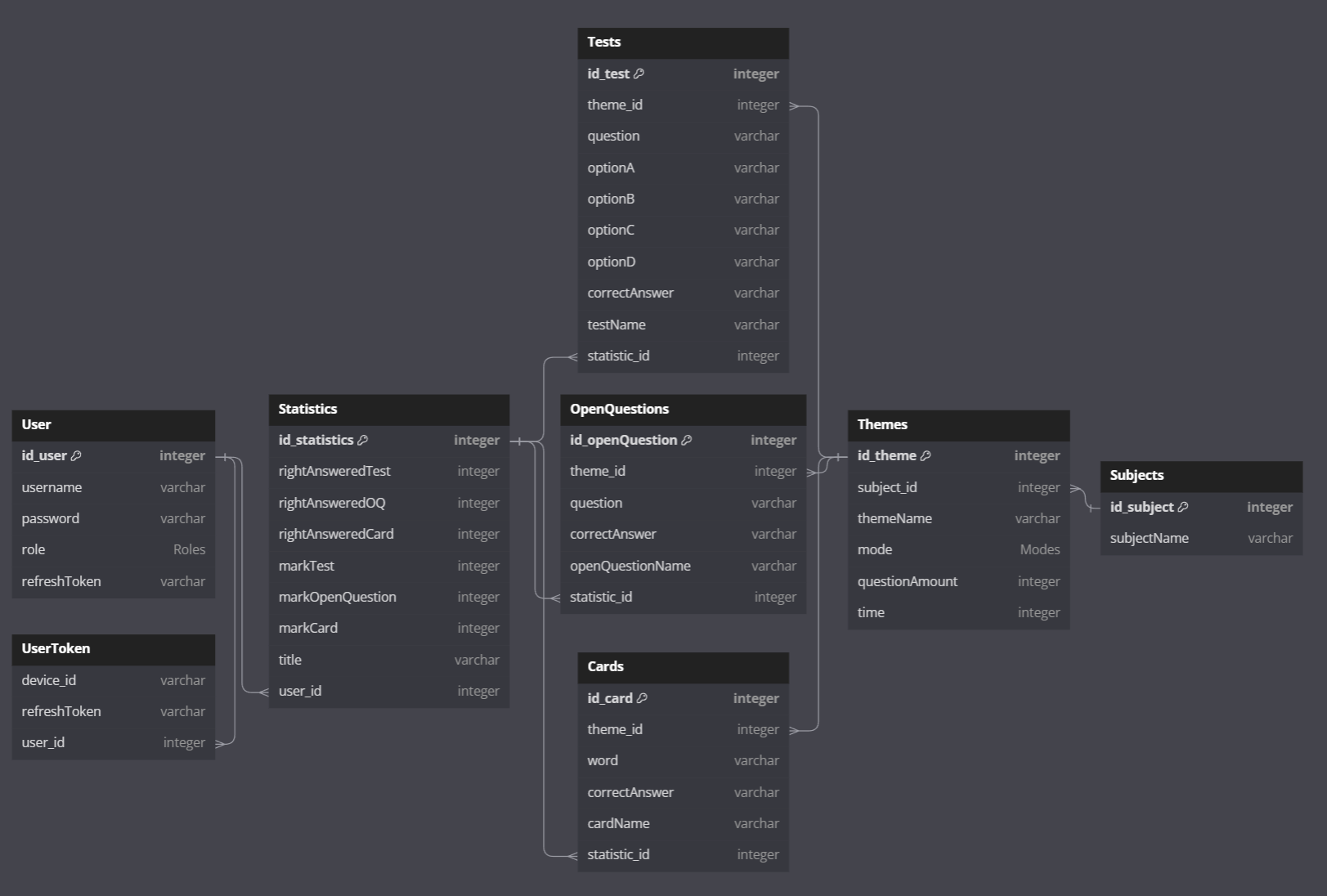


Рисунок 1.1 – Структура базы данных

Следующим шагом является разработка серверной части.

Приложение будет разрабатывалось 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.

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 1.2.

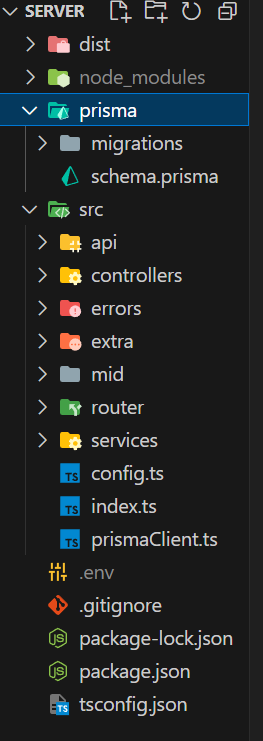


Рисунок 1.2 – Структура проекта

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 1.3 показано распределение интерфеймов, а таперь рассмотрим в пример интерфейс для сождания пользователя:

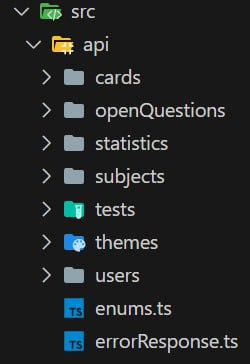


Рисунок 1.3 – Структура папки api

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 1.1 – Структура интерфейса для создания пользователя

Папка *«сontrollers»* отвечает за обработку системных событий, и при этом он не относится к интерфейсу пользователя. Controller определяет методы для выполнения системных операций. В ней находятся контроллеры для работы с каждой сущностью. Одним из условий проекта было построения приложения, который работает в стиле REST. REST из себя представляет архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

REST-архитектура предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером:

* Get;
* post;
* put;
* delete.

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

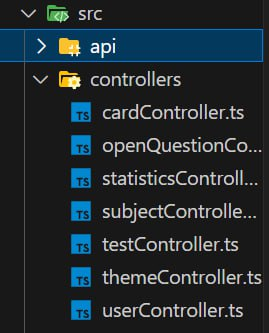


Рисунок 1.4 – Структура папки controller

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 1.2 – 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.

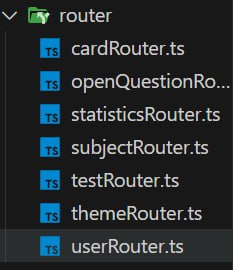


Рисунок 1.5 – Структура папки router

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 1.3 – 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.)

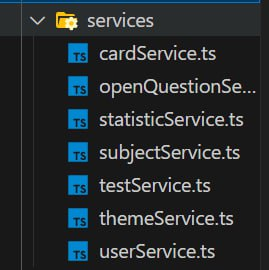


Рисунок 1.6 – Структура папки service

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 1.4 – 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, «Subjects» и «Users».

1.2 Администрирование проекта

В современном мире существует большое количество сред и оперативных систем, которые имеют свою инфраструктуру. Приложения, созданные на одной ОС, вряд ли заработают на другой ОС. Так же, чтобы запустить приложение на той же ОС, на какой создавалось приложение, требуется установить вспомогательные среды разработки, базы данных. Это не эффективно. На мой взгляд, наиболее быстрый и практичный способ является использование Docker.

Docker в настоящее время является одним из самых популярных средств для автоматизации развертывания и управления приложениями в различных средах, например, в операционных системах MacOS, Windows и Linux. Docker одинаково работает независимо от окружающей его среды. Что бы корректно развернуть приложение в контейнере Docker, необходимо написать скрипты развертывания.

Само веб-приложение разработано на базе операционной системы Windows 10. На ней же и будет продолжен процесс настройки администрирования веб-приложения «МедТестЧек».

Самым удобным способом, благодаря которому можно осуществить поддержку «контейнеризации» приложения, является возможность создания докер-файлов, содержащих в себе базовую конфигурацию. В случае серверной части приложения, Visual Studio имеет встроенную поддержку Docker. Для создания Dockerfile, нужно выбрать «Решение», затем «Добавить» и «Поддержка Docker». По окончанию будет создан Dockerfile со следующим содержимым, приведенным на листинге 1.5, который необходим для запуска нашего контейнера с приложением.

FROM node:21-alpine3.18

WORKDIR /app

COPY . .

WORKDIR /app/server

RUN npm i

RUN npx prisma generate

RUN npm run compile

WORKDIR /app/client

RUN npm i

WORKDIR /app

RUN chmod +x startup.sh

CMD ["/bin/sh", "startup.sh"]

Листинг 1.5 – Dockerfile серверной части проекта

Здесь можно выделить следующий порядок действий:

* В начале мы качаем образ (image) из DockerHub самой платформы node.js и делаем публикацию папки app;
* Затем мы устанавливаем рабочий каталог серера внутри контейнера, выполняем команду npm i внутри контейнера, генерируем код, связанный с базой данных, используя инструмент Prisma, а затем компилируем исходный код приложения;
* После мы устанавливаем рабочий каталог клиента внутри контейнера, выполняем команду npm i внутри контейнера;
* И последнее мы устанавливаем права на выполнение для файла startup.sh внутри контейнера и задаем команду, которая будет выполнена при запуске контейнера.

1.3 Начало работы

Для работы с контейнером Docker нужна сама среда Docker [2]. Для этого нужно с официального сайта скачать установщик Docker Desktop. Процесс установки показан на рисунке 1.7.

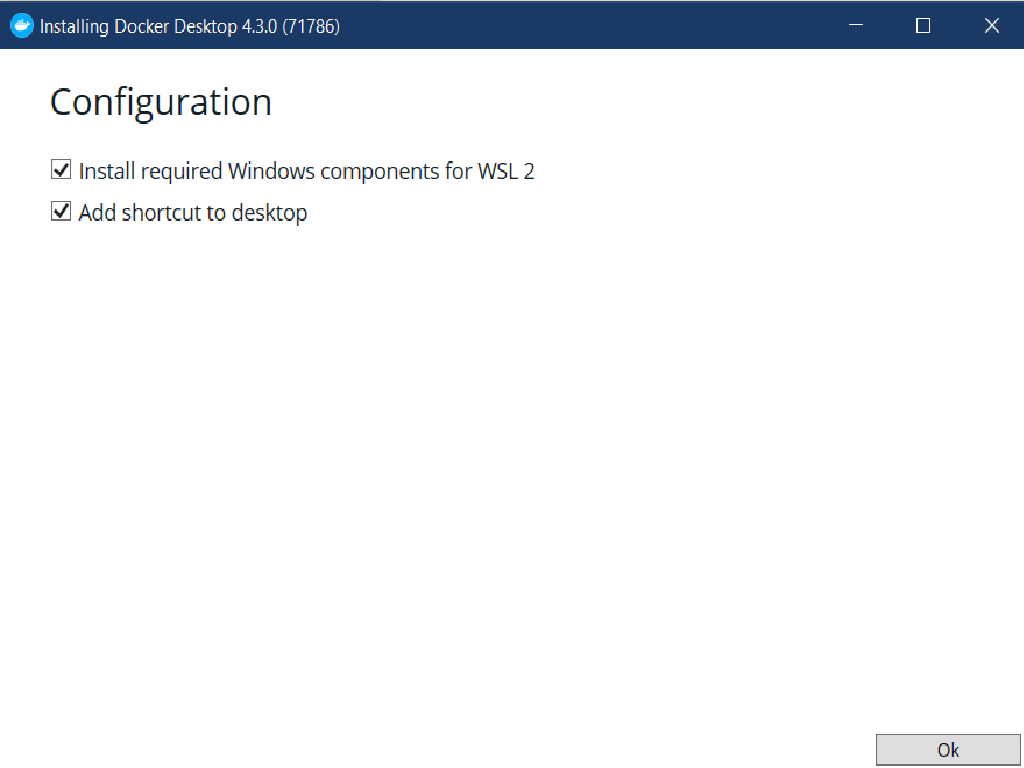


Рисунок 1.7 – Установка Docker Desktop

Далее система предлагает обновить подсистему Windows Subsystem for Linux.

Windows Subsystem for Linux (WSL) – подсистема ОС Windows 10, позволяющая разработчикам, тестировщикам запускать нативные приложения Linux, писать скрипты, выполнять команды непосредственно из Windows [3]. В обновлённой Windows 10 (2004) появилась вторая версия WSL, в которой используется полноценное ядро Linux с возможностью запуска приложений и контейнеров Docker, реализована высокая скорость загрузки, небольшой объем потребляемых ресурсов, управление в фоновом режиме, обновление ядра. Таким образом вы сможете запускать ELF64 программы, которые могут получать доступ к файловой системе Windows. WSL – это, в первую очередь, средство для разработчиков, особенно веб-разработчиков, работающих над проектами с открытым кодом или развертывания в среде Linux Server. WSL предназначен для всех, кто любит использовать Bash, общие средства Linux и Linux – первые платформы (Ruby, Python и т. д.) но также с помощью Windows средств повышения производительности.

После установки запускаем приложение. Чтобы приложение работало корректно, необходимо убедиться, что имеется достаточно свободного места на жестком диске и есть доступ к интернету. По умолчанию свободного места должно быть не менее 10гб.

После этого мы откроем PowerShell и пропишем команду «docker-compose up -d», представленную на рисунке 1.8.

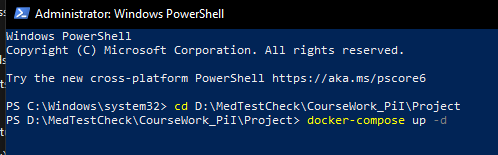


Рисунок 1.8 – Использование специальной команды

Команда docker-compose up -d используется для запуска контейнеров на основе файлов конфигурации Docker Compose.

Docker Compose - это инструмент, который позволяет определить и управлять множеством контейнеров в вашем приложении с помощью одного файла конфигурации. Файл конфигурации Docker Compose, обычно называемый docker-compose.yml, описывает сервисы (контейнеры), их зависимости, настройки и другие параметры.

Когда вы выполняете команду docker-compose up -d, Docker Compose читает файл конфигурации, создает и запускает контейнеры на основе описанных сервисов. Флаг -d (или --detach) указывает Docker Compose запустить контейнеры в фоновом режиме (detached mode), что означает, что они будут работать в фоновом режиме и не будут выводить свои логи в текущий терминал.

В результате выполнения команды docker-compose up -d, ваши контейнеры будут запущены и будут работать в фоновом режиме в соответствии с определенными настройками в файле docker-compose.yml.

Результат скачивания можно увидеть на рисунке 1.9 и 1.10.

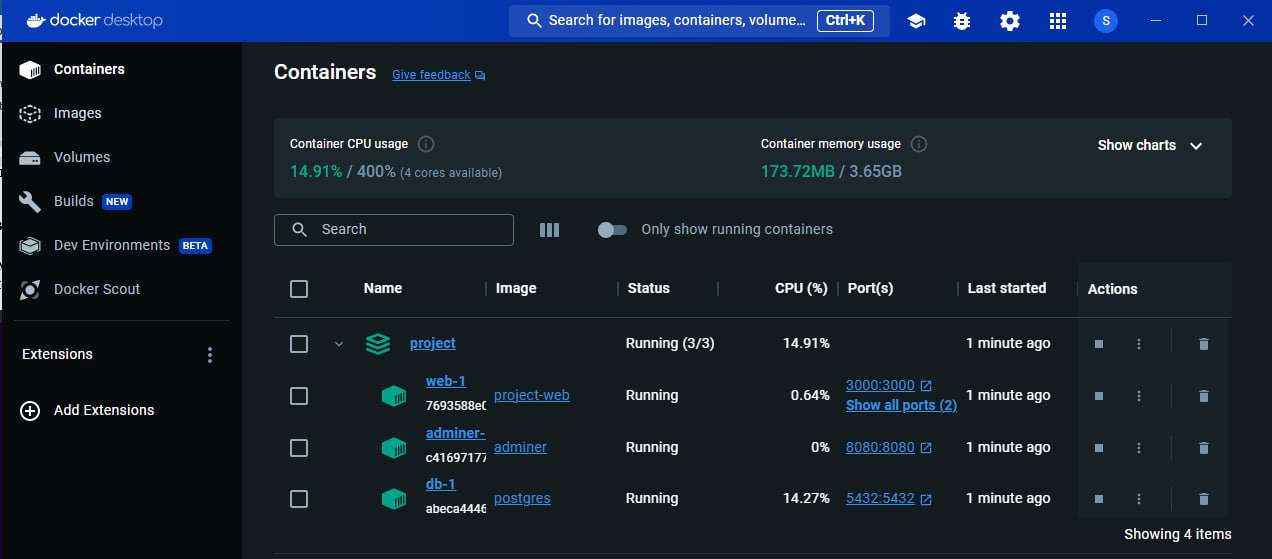


Рисунок 1.9 – Контейнеры в Docker Desktop

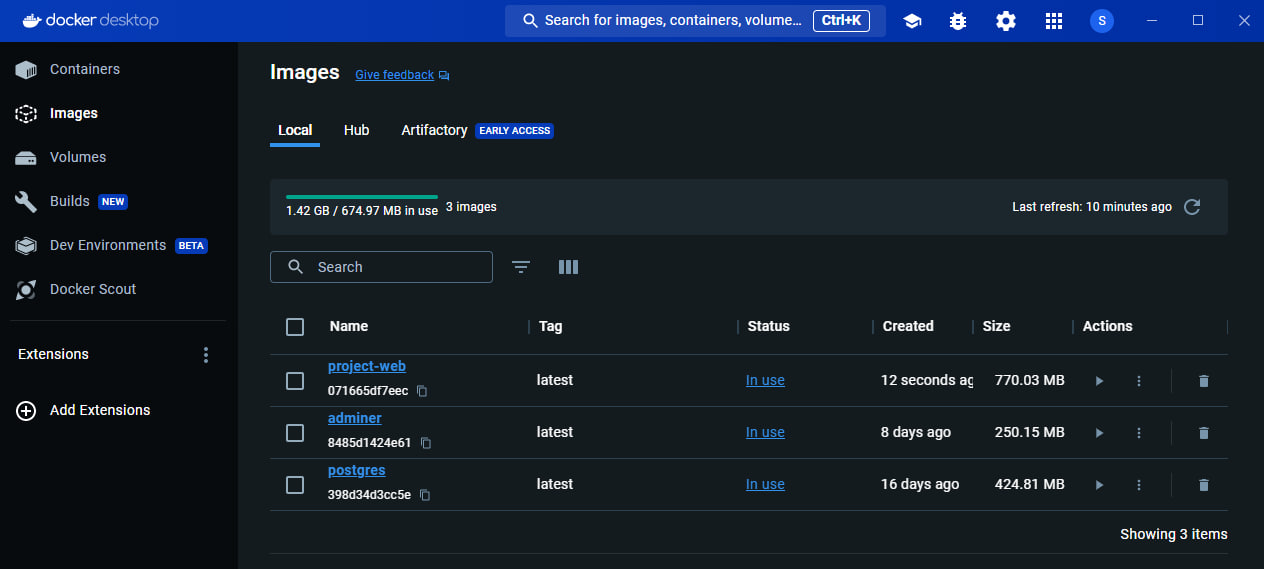


Рисунок 1.10 – Images в Docker Desktop

И так, у нас существует два контейнера, которые имеют порты, которые мы задали. Контейнеры были успешно установлены. Использование Docker позволяет быстрее и эффективнее доставлять или перемещать код, стандартизирует выполняемые приложениями операции, оптимизируя использование ресурсов. Благодаря Docker пользователи получают объект, который с высокой надежностью можно запускать на любой платформе. Простой и понятный синтаксис Docker обеспечивает полный контроль над выполняемыми операциями.

Из этого можно сделать вывод, что мы развернули проект на базе Docker успешно, осталось только проверить работоспособность приложения.

**2 Проверка работоспособности приложения**

Первым делом необходимо запустить приложение. Нас встречает авторизация, на которой можно войти в приложение. Результат страницы показан на рисунке 2.1.

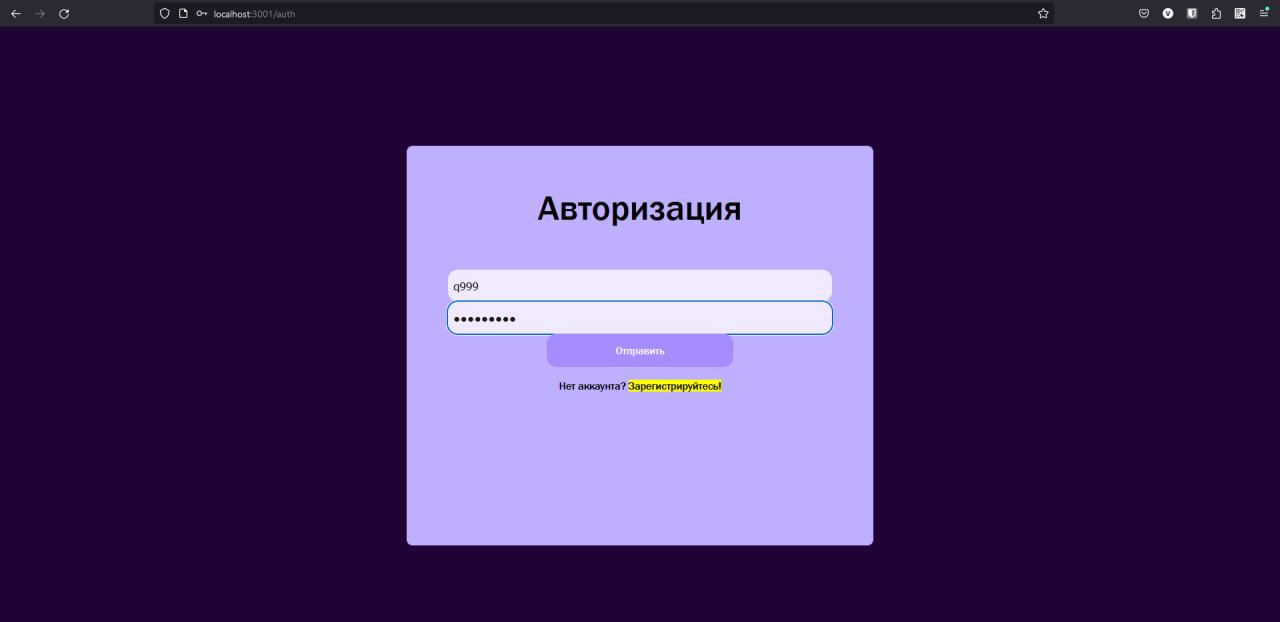


Рисунок 2.1 – Страница авторизации

Администратор зоздается заранее, поэтому за него можно только зайти.

Перейдем на страницу со списком пользователей нашего приложения, которая показана на рисунке 2.2.

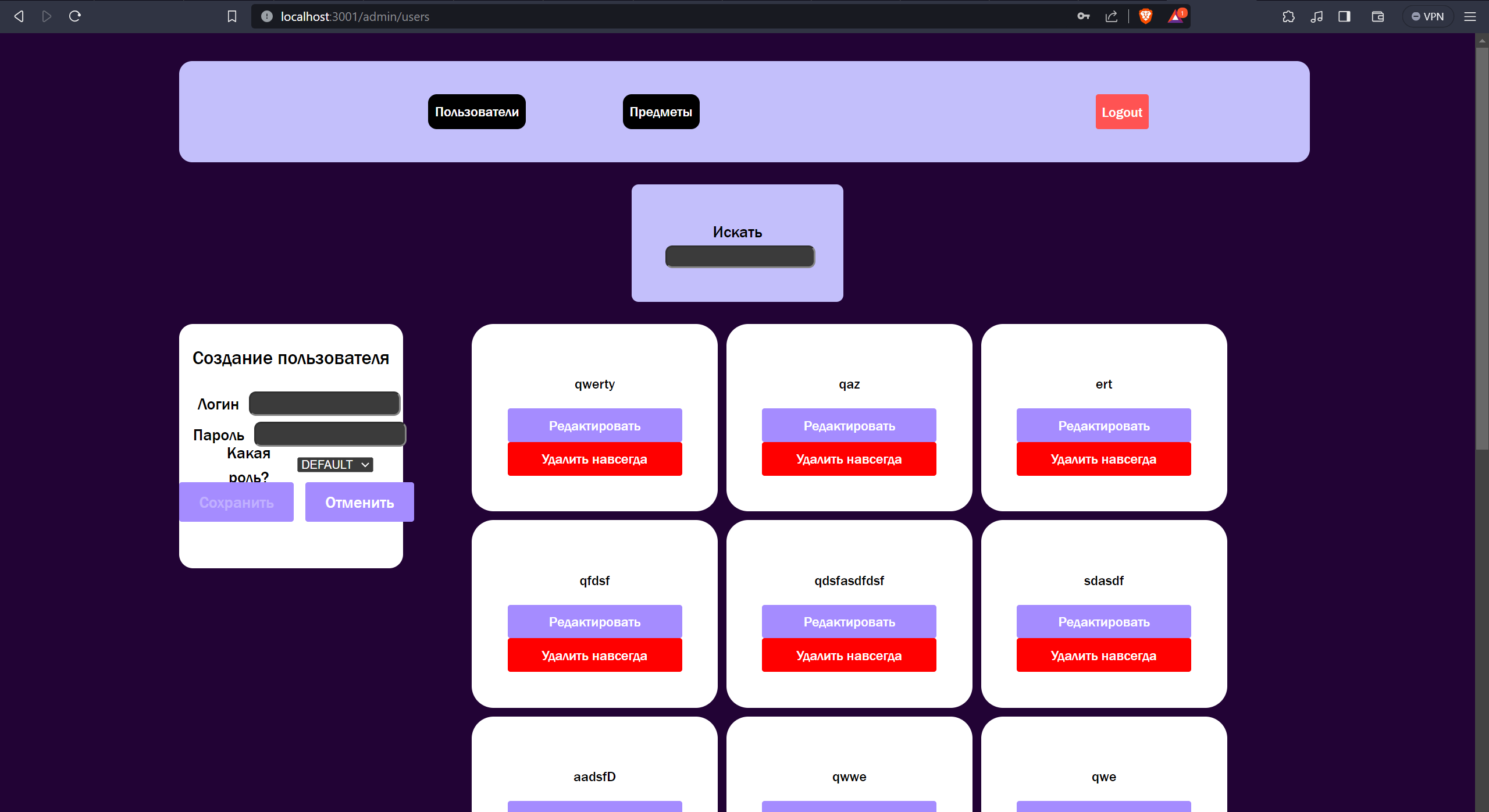


Рисунок 2.2 – Страница пользователей

Следующим шагом переходим на страницу «Предметы», в которой выводится информация об учебных предметах. Также мы можем посмотреть детали созданных предметов, в которых выводится информация о предмете и возможность редактировать. Также мы можем удалить предметы, как показано на рисунке 2.3.

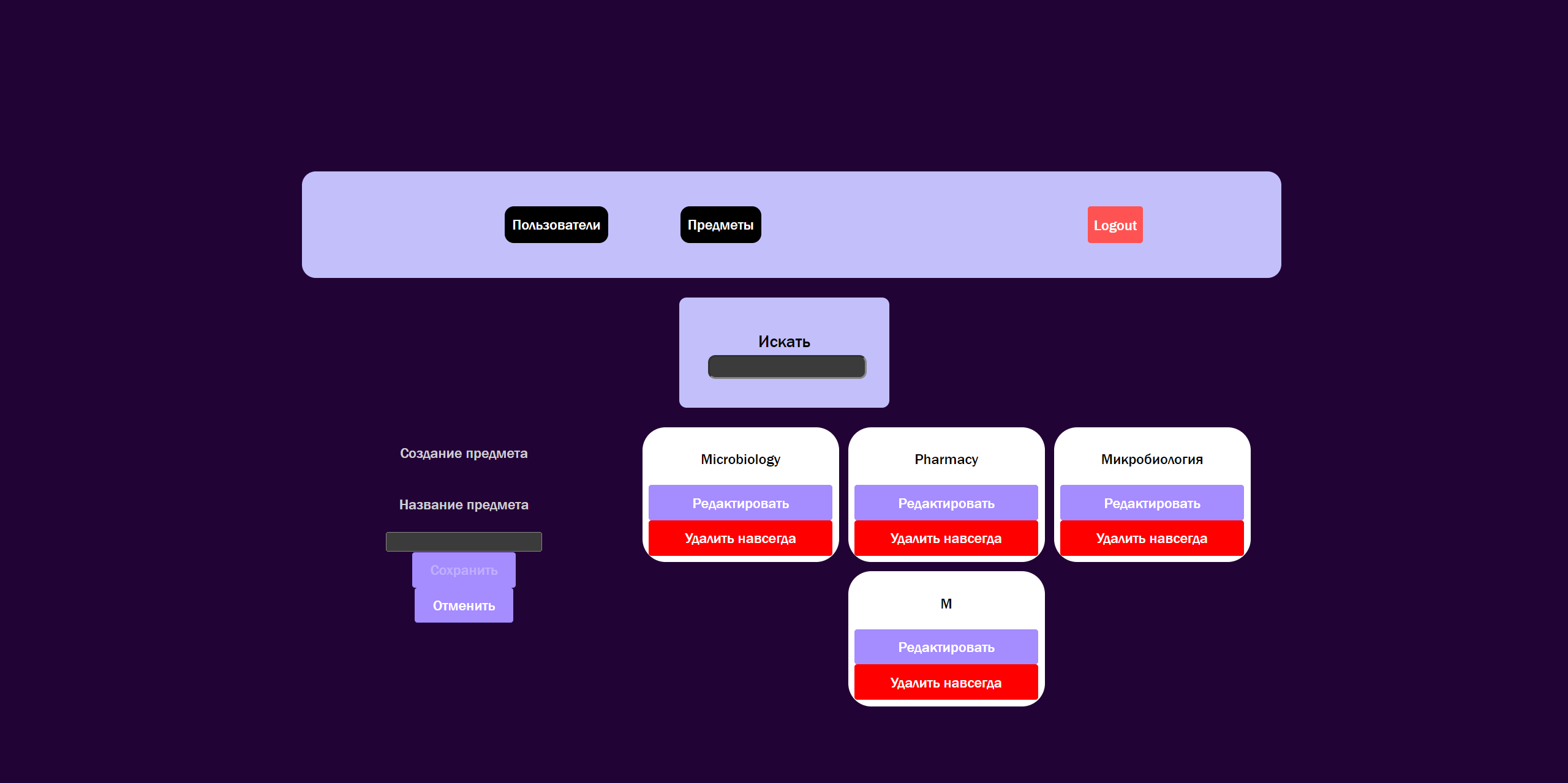


Рисунок 2.3 – Страница предметов

Далее мы рассмотрим роль преподавателя, у которого тоже не мало возможностей. Авторизируемся как преподаватель и переходим на страницу «Темы», на которой мы можем увидеть список тем предметов и вид закрепления знаний. Также мы можем редактировать и удалить предметы. – Страница темы показан на рисунке 2.4.

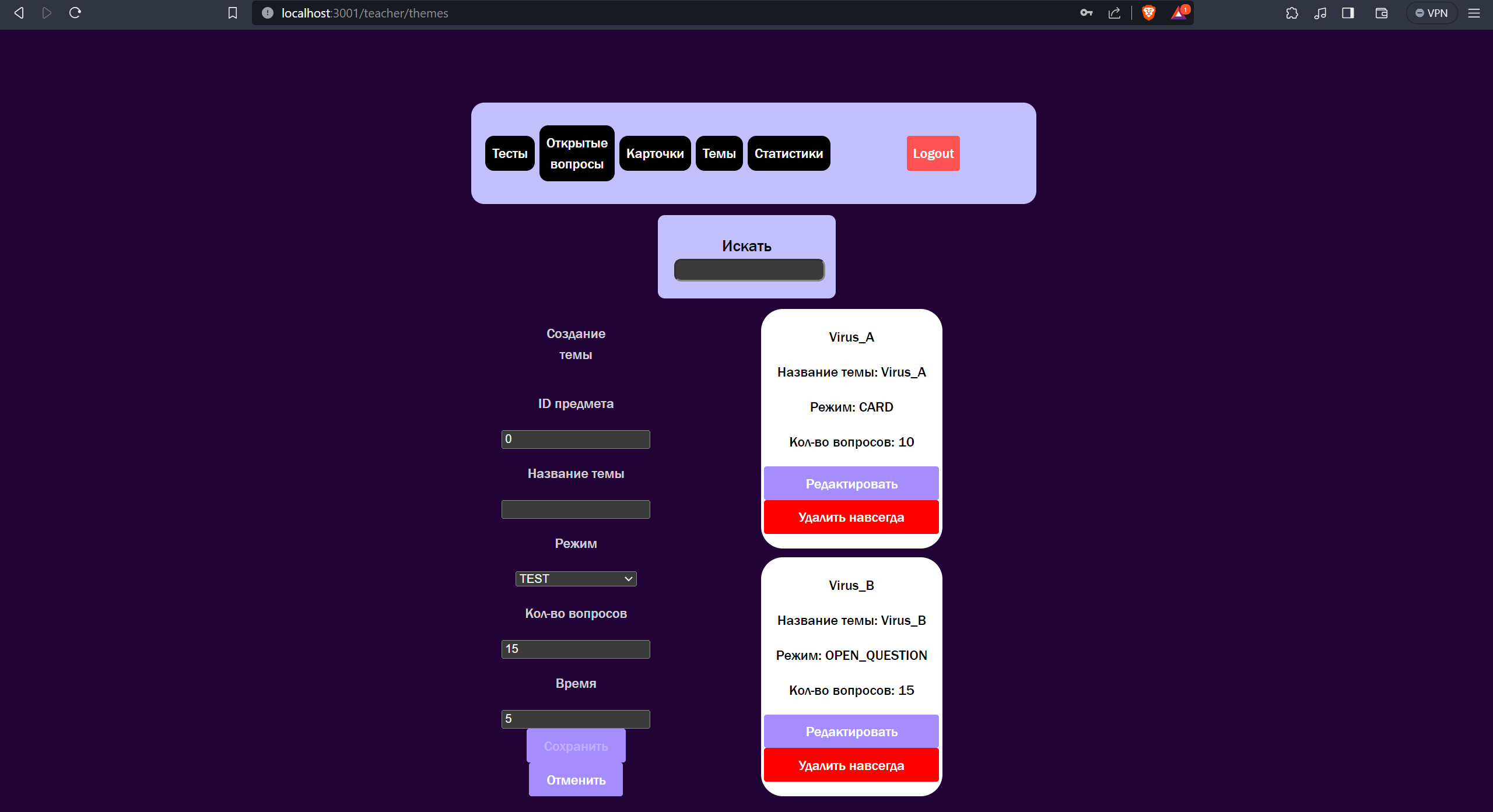


Рисунок 2.4 – Страница темы

Следующая представленная страница – страница «Карточек». Страница посвящена карточкам, и мы можем их создавать, редактироввть и удалять, как показано на рисунке 2.5.

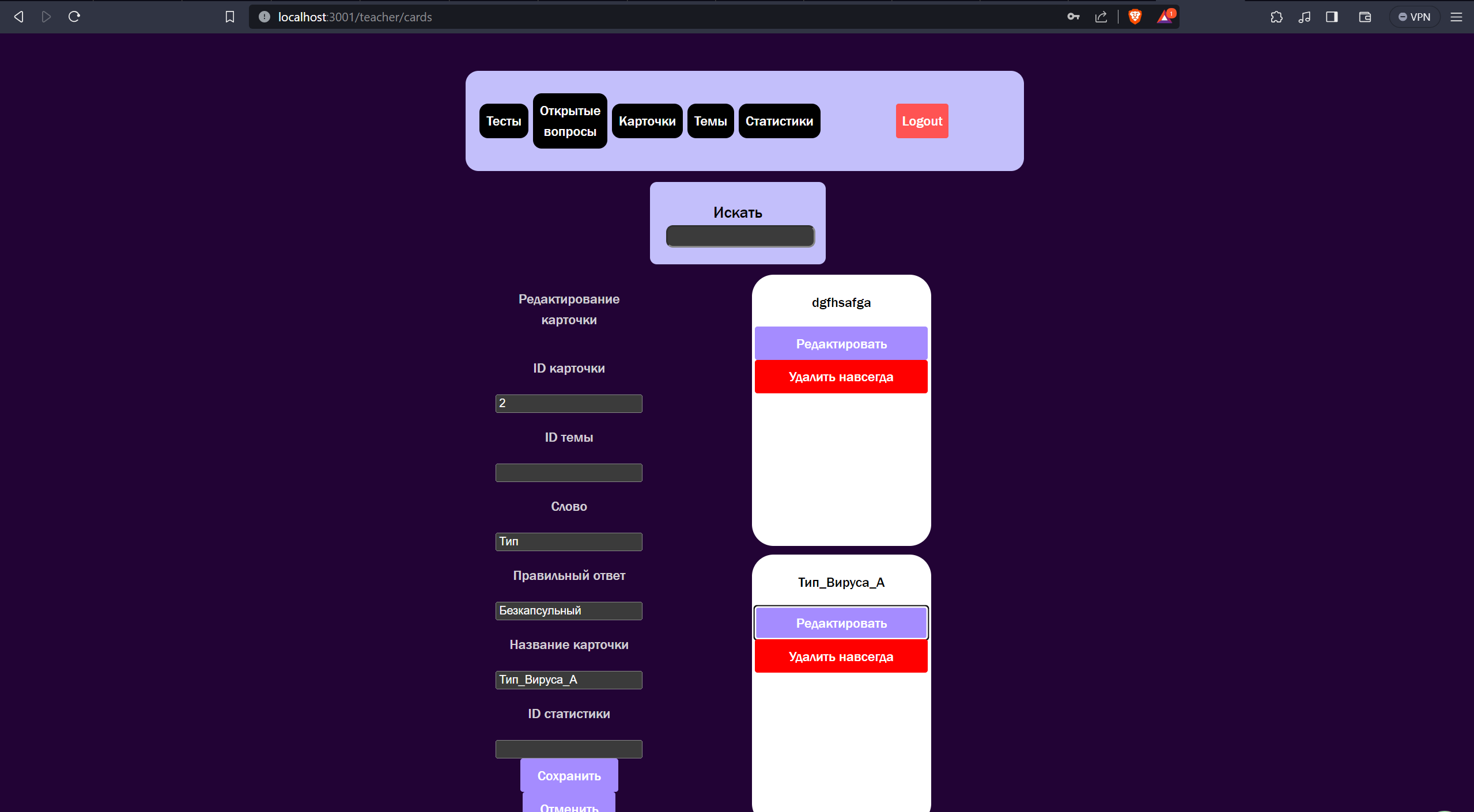


Рисунок 2.5 – Страница карточек

В страницах «Тесты» и «Открытые вопросы» то же самое.

Таким образом, администратор – это наиболее уполномоченный пользователь. В каждом приложение необходимо наличие такого пользователя для управления и администрирования приложения.

**Заключение**

В ходе выполнения курсовой работы 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 серверной части использовалась платформа NodeJS. NodeJS может быть развернута на основных популярных операционных системах: Windows, Mac OS, Linux. Также для реализации приложения использовался паттерн проектирования MVC, который представляем из себя совукупность использования модели, представления и контроллера. В результате была разработан веб-сервис по архитектуре REST, который обрабатывает клиентские запросы и делает запросы к базе данных нашего web-приложения.

Была разработана реляционная база данных для веб-приложения, использующая экземпляр Postgres, имеющая в своем наборе 8 таблиц, которые содержат в себе все необходимые данные для успешного использования.

Были созданы дополнительные конфигурационные файлы, позволяющие быстро и легко администрировать приложением и расширять его функционал, например, для его дальнейшней проектировки и усовершенствования.

В результате проведенной работы, был создана документация к проекту, который удовлетворяет потребностям пользователей и корректным образом выполняет все необходимые функции. При этом приложение обладает понятным и удобным интерфейсом, который будет понятен и доступен пользователю любого уровня.

В данной курсовой работе мы познакомились с Docker, подсистемой Windows для Linux, создавали Images и контейнеры.

Также я получил как опыт проектирования, так и опыт разработки приложения. Приобрёл опыт работы с большим стеком технологий и изучил новые.

Опыт развертывания приложений, полученный во время выполнения курсовой работы, считаю немаловажным дополнением ко всем приобретённым навыкам проектирования и построения приложения.

**Список используемых источников**

1. Node.js [Электронный ресурс] – Режим доступа: https://metanit.com/web/nodejs/1.1.php– Дата доступа: 17.11.2023.

2. Руководство по Docker [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/ruvds/blog/450312/– Дата доступа: 26.11.2023.

3. Основные команды для WSL [Электронный ресурс] – Режим доступа: https://learn.microsoft.com/ru-ru/windows/wsl/basic-commands – Дата доступа: 27.11.2022.

4. PosgreSQL [Электронный ресурс] – Режим доступа: https://www.postgresql.org/– Дата доступа: 28.11.2023.

5. Docker Hub [Электронный ресурс] – Режим доступа: https://hub.docker.com/– Дата доступа: 04.12.2023.