МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА»

ИНСТИТУТ ИНФОРМАТИКИ И КИБЕРНЕТИКИ

КАФЕДРА ТЕХНИЧЕСКОЙ КИБЕРНЕТИКИ

ОТЧЕТ ПО ЛАБОРАТОРНЫМ РАБОТАМ

Тема: «Проектирование приложения»

Дисциплина: «Технологии сетевого программирования»

Выполнил: Якухин И.В., гр.6303

Проверил: Кашапов А.И.

Самара 2025

СОДЕРЖАНИЕ

[Введение 3](#_Toc199556885)

[1 Структура приложения 4](#_Toc199556886)

[1.1 Компоненты приложения 4](#_Toc199556887)

[1.2 Docker-compose 4](#_Toc199556888)

[2 Описание API 5](#_Toc199556889)

[3 Тестирование API 8](#_Toc199556890)

[4 Аутентификация и авторизация 11](#_Toc199556891)

[4.1 JWT-аутентификация 11](#_Toc199556892)

[5 Пользовательский интерфейс 12](#_Toc199556893)

[6 Структура контейнеризации и настройки окружения 13](#_Toc199556894)

[6.1 Архитектура контейнеров 13](#_Toc199556895)

[6.2 Docker-compose 13](#_Toc199556896)

[7 Демонстрация работы приложения в контейнерах 14](#_Toc199556897)

[Приложение А 15](#_Toc199556898)

[Ссылка на GitHub с проектом 15](#_Toc199556899)

[Приложение Б 16](#_Toc199556900)

[Файл docker-compose.yml 16](#_Toc199556901)

[Dockerfile клиентской части 16](#_Toc199556902)

[Dockerfile серверной части 17](#_Toc199556903)

[Маршрутизация в App.jsx 17](#_Toc199556904)

Введение

Cardiogram – приложение с увлекательной игрой для запоминания английских слов с помощью карточек. Стек технологий приложения: Python, Django, React, PostgreSQL. Все компоненты конечной системы контейнеризованы с помощью Docker для обеспечения изоляции окружения, удобства развертывания и масштабирования.

Учебный проект разработан с целью демонстрации навыков работы с базой данных и инструментами серверной и клиентской разработки, проектирования структуры приложений и API, контейнеризации.

Ссылка на GitHub страницу со всеми файлами проекта можно найти в приложении А.

1. Структура приложения
   1. Компоненты приложения

Приложение разделено на 3 компонента: база данных, серверное приложение (backend), клиентское приложение (frontend).

1. **База данных**
   * Хранит все данные приложения: пользователей, колоды, карточки, прогресс.
2. **Серверное приложение**
   * Обрабатывает запросы клиента;
   * Выполняет внутреннюю логику приложения (регистрация, аутентификация, сохранение прогресса);
   * Взаимодействует с базой данных;
   * Возвращает данные клиенту.
3. **Клиентское приложение**
   * Отображает интерфейс, с которым взаимодействует пользователь;
   * Отправляет запросы на сервер для получения или записи данных.
   1. Docker-compose

Каждый компонент приложения был упакован в контейнер и их совокупность управляется через файл docker-compose.yml.

Особенности работы компонентов, прописанные в docker-compose:

* Строгая последовательность запуска компонентов (база данных, сервер, клиент) позволяет избежать ситуаций, когда клиент отправляет запрос на еще не запущенный сервер;
* При остановке контейнера он снова запускается, как только это становится возможным;
* Получение переменных из .env файла через docker-compose повышает безопасность кода.

Описание API

Для реализации API методов в Django приложении был создан отдельный модуль api. С использованием классов из библиотеки Django-rest-framework были созданы сериализаторы для корректной обработки данных, view-классы, регламентирующие формат запросов и ответов и порядок обработки данных, и файл c путями маршрутизации запросов. Код из файлов “serializers.py”, “views.py”, “urls.py” приведены в приложении Б.

Структура API, включая методы, URL, параметры, форматы запросов и ответов, описана в таблице 1.

Таблица 1 – Описание API

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Действие | Метод | URL | Параметры | Формат запроса | Формат ответа |
| Получение информации о карточке | GET | card/<id> | Id – номер карточки в таблице базы данных |  | {  “id”: 23,  “front\_text”: “forest”,  “back\_text”: “Лес”,  “example\_usage”: “Forest is beautiful! (Лес прекрасен!)”  } |
| Получение информации о колодах | GET | deck/ |  |  | [{  “id”: 4,  “name”: “Nature”,  “description”: “Колода содержит слова, связанные с темой природы.”,  “cover”: “http://127.0.0.1/deck/forest.png”  },  {  …  }] |
| Получение из колоды всех карточек, которых нет у пользователя | GET | deck/<id> | Id – номер колоды в таблице базы данных |  | [{  “id”: 23,  “front\_text”: “forest”,  “back\_text”: “Лес”,  “example\_usage”: “Forest is beautiful! (Лес прекрасен!)”  },  {  “id”: 23,  “front\_text”: “nature”,  “back\_text”: “природа”,  “example\_usage”: “Nature is beautiful! (Природа прекрасна!)”  },…] |
| Регистрация | POST | auth/register | username,  first\_name,  last\_name,  email,  password | {“username”: “Werd”,  “first\_name”: “Иван”,  “last\_name”: “Якухин”,  “email": “werb32@mail.ru”,  “password”: “1234”  } | {“message”: “registration successful”} |
| Авторизация | POST | auth/login | username,  password | {“username”: “Werd”,  “password”: “1234”  } | {“message”: “login successful”} |
| Смена пароля | POST | auth/change-password | old\_password,  new-password | {  “old\_password”: “1234”,  “new\_password”: “1111”  } | {“message”: “password changed successful”} |
| Выход из системы | POST | auth/logout |  |  | {“message”: “logout successful”} |
| Изменение личных данных | POST | auth/edit | username,  first\_name,  last\_name,  email | {“username”: “Werd”,  “first\_name”: “Иван”,  “last\_name”: “Якухин”,  “email": “werb32@mail.ru”  } | {“message”: “Profile editing completed successfully!”} |
| Получение данных о прогрессе пользователя | GET | progress/ |  |  | [{  “user”: 23,  “card”: 34,  “attempts”: 23,  “successful\_attempts”: 10  },  {  “user”: 23,  “card”: 32,  “attempts”: 21,  “successful\_attempts”: 5  },…] |
| Добавление карточки пользователю | POST | progress/ | card | {“card”: 23} | {“message”: “Card added to your deck successfully!  ”} |
| Получение профиля пользователя | GET | profile/ |  |  | {“username” : “Werd”,  “first\_name”: “Иван”,  “last\_name”: “Якухин”,  “email": “werb32@mail.ru”  } |
| Пользователь правильно выбрал карточку | PATCH | profile/right | card | {“card”: 23} | {“message”: “the card information has been updated”} |
| Пользователь неправильно выбрал карточку | PATCH | profile/wrong | card | {“card”: 23} | {“message”: “the card information has been updated”} |
| Удаление аккаунта пользователя | DELETE | delete/ |  |  | {'message': 'Account deleted successfully!'} |

1. Тестирование API

Каждый метод созданного API был протестирован в приложении Postman в режиме аутентифицированного и не аутентифицированного пользователя для проверки корректности обработки запросов и защиты эндпоинтов.

На рисунках 1-5 представлены скриншоты успешно и неуспешно завершенных запросов.

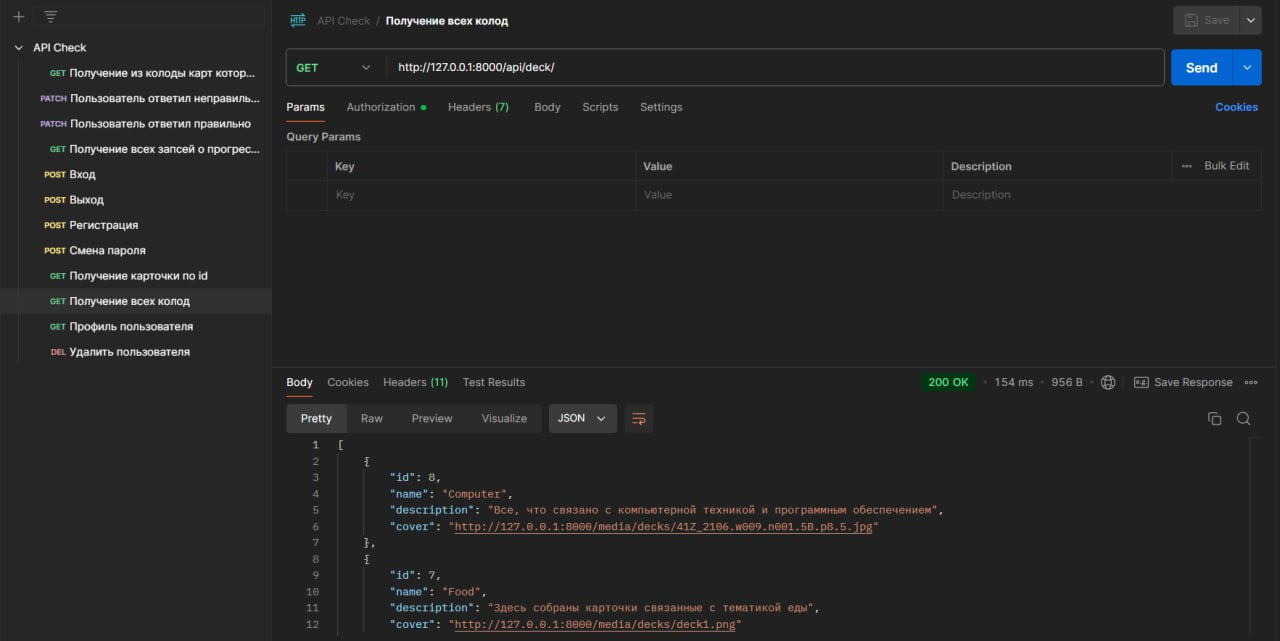


Рисунок 1 – Скриншот успешного запроса информации о колодах

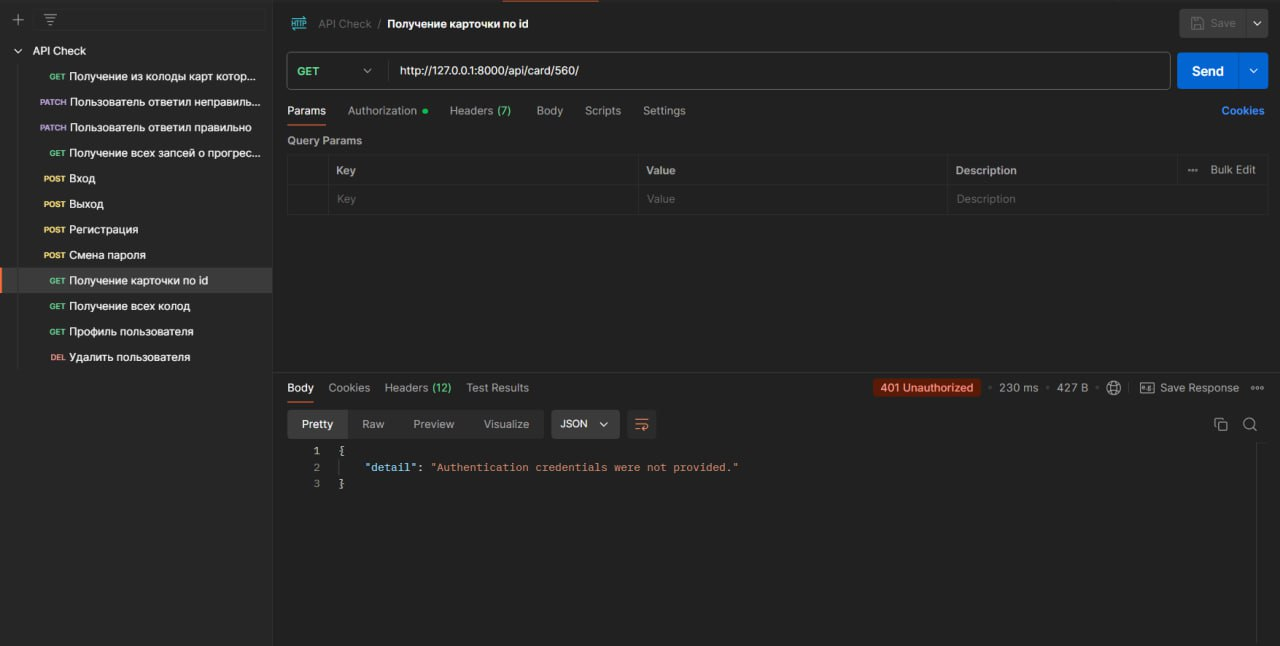


Рисунок 2 – Ошибка: пользователь не прошел аутентификацию

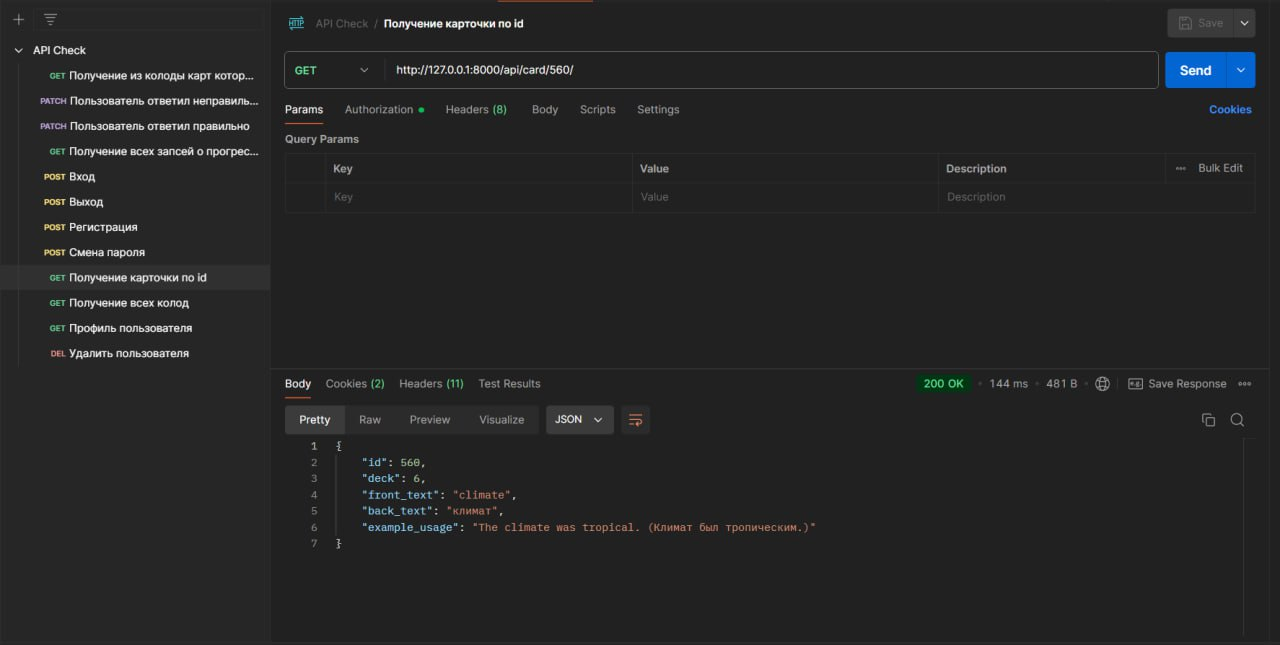


Рисунок 3 – Успешный запрос информации о карточке

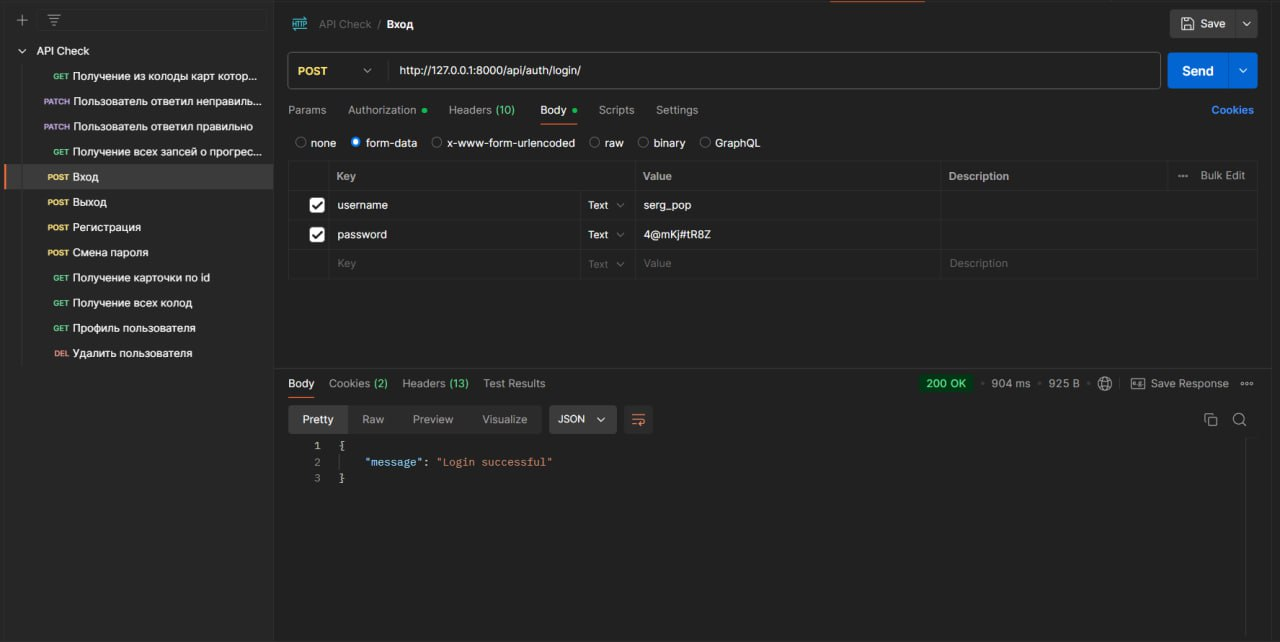


Рисунок 4 – Успешный запрос на вход в систему

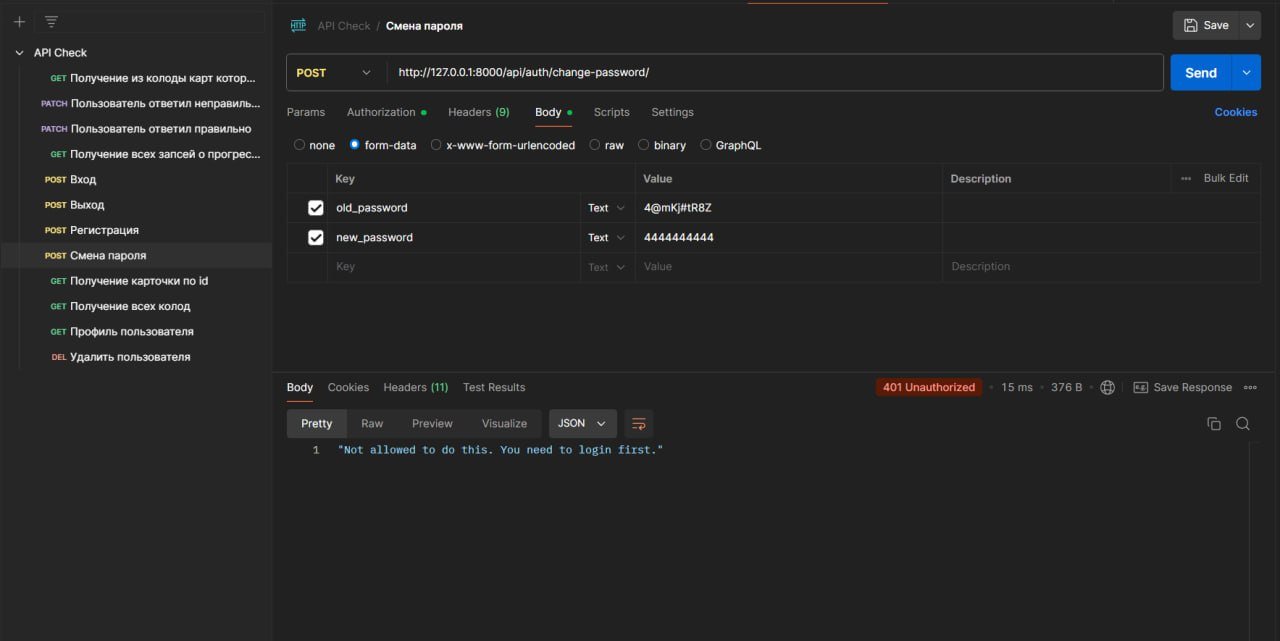


Рисунок 5 – Ошибка: не аутентифицированный пользователь не может менять пароль

1. Аутентификация и авторизация

Серверное приложение реализует аутентификацию с использованием JWT токенов для управления доступом пользователей. В качестве механизма авторизации выступает разрешение isAuthenticated позволяющее доступ к определенным эндпоинтам только аутентифицированным пользователям.

* 1. JWT токены

При входе в систему пользователь передает свои учетные данные с запросом на сервер. В случае успешного выполнения запроса сервер запишет access и refresh токены в Cookie файлы на стороне клиента, чтобы тот, посылая следующий запрос, был аутентифицирован по токену из Cookie. Для проверки валидности JWT токенов и обновления access токена был написан middleware класс RefreshAccessTokenMiddleware, код которого представлен в приложении Б. JWT токены как в API запросах, так и в middleware записываются в HTTPOnly Cookie файлы.

1. Пользовательский интерфейс

Пользовательский интерфейс приложения разделен на несколько страниц:

* Главная страница – доступна всем, знакомит пользователя, с тем, как устроен сайт;
* Профиль – страница, где авторизованный пользователь может увидеть свою личную информацию и ознакомиться со статистикой своего обучения;
* Колоды – доступна всем, позволяет ознакомиться c темами колод иностранных слов;
* Карты в колоде, карты пользователя, страница обучения – доступны авторизованным пользователям, позволяют увидеть все карты, представленные в приложении.

Клиентское приложение создано с помощью JavaScript библиотеки React. Для маршрутизации между страницами в клиентском приложении используется библиотека react-router-dom, а для запросов к серверу используется axios с учетом настроек CORS политики на сервере. Файл App.jsx, реализующий маршрутизацию по страницам приведен в приложении Б.

HTML шаблоны приложения были преобразованы в React компоненты. Стили из .css файлов были перенесены в .module.css файлы и правильно подключены в React. Следующим шагом было налажено взаимодействие клиентского приложения с серверным. В React компонентах запросы к серверу посылаются с помощью библиотеки axios. Данные с сервера попадают в клиентское приложение и динамически отрисовываются на страницах. Пример React компонента с запросом к серверу и динамическим обновлением страницы приведен в приложении Б. На рисунке 3 представлена страница с изображением и текстом, полученными от сервера.

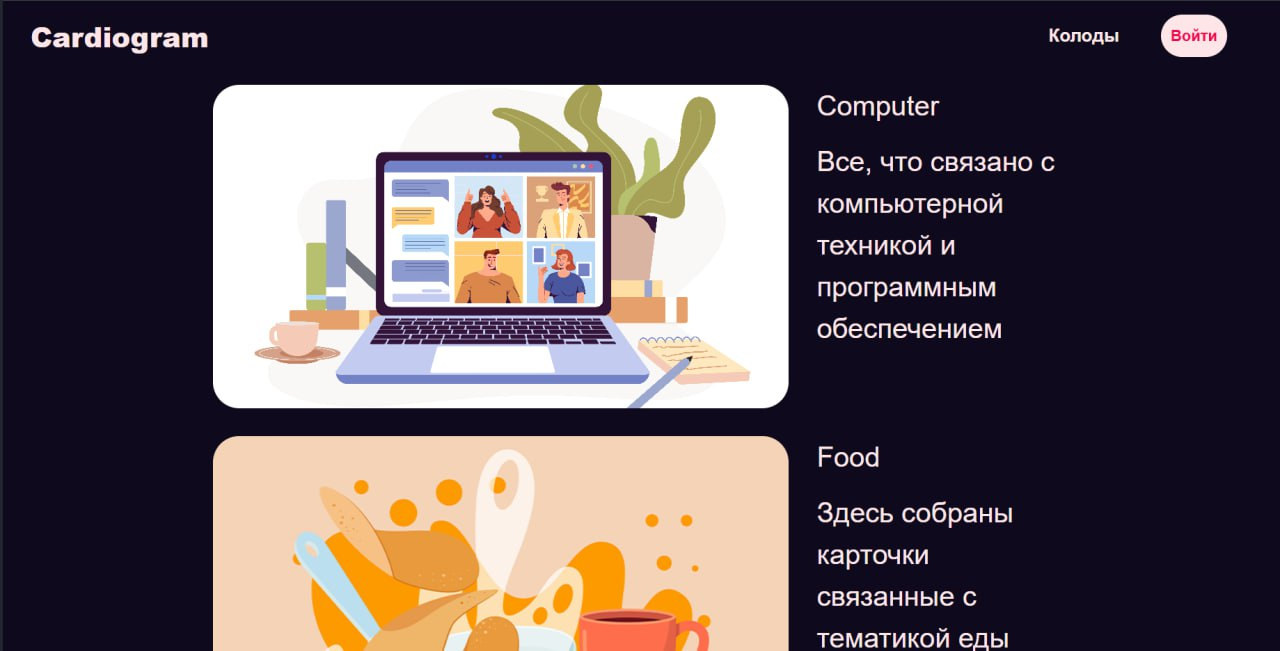


Рисунок 6 – Скриншот страницы с колодами, на которой данные загружаются с сервера

1. Структура контейнеризации и настройки окружения

Все компоненты конечной системы упакованы в Docker контейнеры для обеспечения изоляции окружения, удобства развертывания и масштабирования.

* 1. Архитектура контейнеров

Архитектура упакованного приложения включает в себя 3 контейнера:

* 1. PostgreSQL – контейнер на основе образа последней версии postgres, в котором развернута база данных. Использует volume postgres\_data:/var/lib/postgresql/data для сохранения данных между перезапусками контейнера.
  2. Backend – контейнер содержит Django приложение собранный на основе образа python версии 3.10. Серверное приложение доступно на порте 8000.
  3. Frontend – контейнер содержит React приложение собранное на основе образа Node.js последней версии и запущенное с помощью образа Nginx последней версии.

Код docker файлов приведен в приложении Б.

* 1. Docker-compose

Для управления всеми контейнерами используется файл docker-compose.yml, который описывает их конфигурацию и связи. В docker-compose описывается на основе каких образов создаются контейнеры, как они будут названы, какие у них будут порты и тома, а также особенности их функционирования. Текст файла docker-compose.yml представлен в приложении Б.

Основные особенности конфигурации контейнеров в docker-compose файле:

* Строгая последовательность запуска компонентов (база данных, сервер, клиент). Это позволяет избежать ситуаций, когда клиент отправляет запрос на еще не запущенный сервер;
* Каждый контейнер постоянно работает и перезапускается в случае сбоя или перезапуска Docker. Это позволяет избежать простоев в работе приложения в случае возникновения критического сбоя;
* Контейнеры доступны друг для друга в единой внутренней сети по именам сервисов. Например, для получения данных из БД backend приложение делает запрос по адресу db:5432 (db – название сервиса с базой данных, 5432 – порт, указанный в .env файле.);
* Переменные среды передаются в контейнеры базы данных и серверного приложения из .env фала, лежащего рядом с docker-compose.yml;
* Контейнер с базой данных использует том postgres\_data для хранения данных между перезапусками контейнера.

1. Демонстрация работы приложения в контейнерах

Упакованное приложение запускается командой docker-compose up, после чего сервер становится доступен по адресу 127.0.0.1:8000, а клиент – 127.0.0.1:80. На рисунках 6-7 показана работа контейнеров и приложения в браузере.

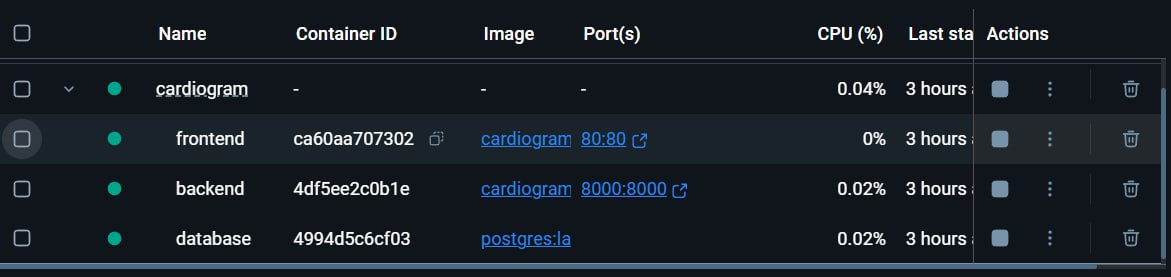


Рисунок 7 – Запущенные контейнеры в приложении Docker Desktop

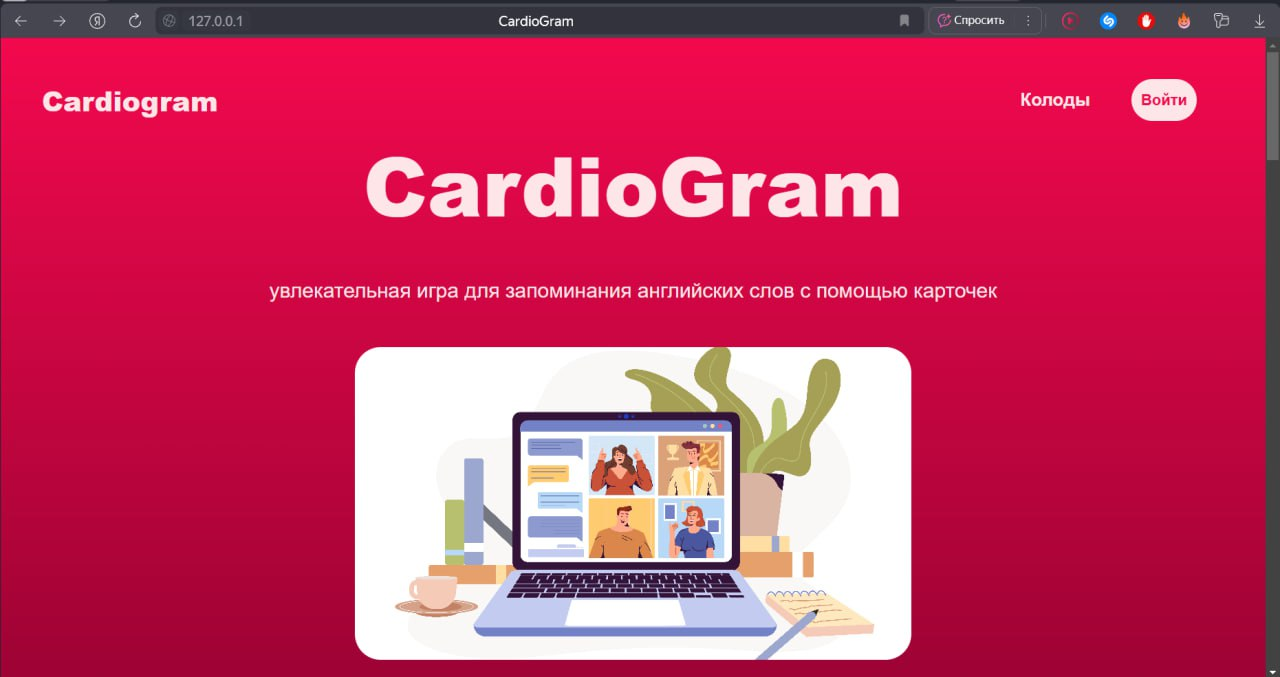


Рисунок 8 – Главная страница приложения в браузере

Приложение А

Ссылка на GitHub с проектом

https://github.com/BruhMano/CardioGram.git

Приложение Б

Код файла “serializers.py”

from rest\_framework.serializers import ModelSerializer, Serializer, CharField, ValidationError

from cards.models import Card

from decks.models import Deck

from users\_progress.models import Progress

from django.contrib.auth import get\_user\_model

class CardSerializer(ModelSerializer):

class Meta:

model = Card

fields = ['id', 'deck', 'front\_text', 'back\_text', 'example\_usage']

class DeckSerializer(ModelSerializer):

class Meta:

model = Deck

fields = ['id', 'name', 'description', 'cover']

class UserProgressSerializer(ModelSerializer):

class Meta:

model = Progress

fields = ['id', 'user', 'card','attempts', 'successful\_attempts']

class UserSerializer(ModelSerializer):

class Meta:

model = get\_user\_model()

fields = ['username', 'email', 'first\_name', 'last\_name']

class ChangePasswordSerializer(Serializer):

old\_password = CharField(required=True)

new\_password = CharField(required=True)

def validate\_old\_password(self, value):

user = self.context['user']

if not user.check\_password(value):

raise ValidationError("Current password is incorrect.")

return value

def validate(self, data):

old\_password = data.get('old\_password')

new\_password = data.get('new\_password')

if old\_password == new\_password:

raise ValidationError("New password must be different from the old password.")

return data

Код файла “views.py”

from api.serializers import CardSerializer, DeckSerializer, UserProgressSerializer, UserSerializer, ChangePasswordSerializer

from rest\_framework.response import Response

from rest\_framework import permissions, generics, views

from cards.models import Card

from decks.models import Deck

from users\_progress.models import Progress

from django.contrib.auth import get\_user\_model

from rest\_framework\_simplejwt.tokens import RefreshToken

from django.contrib.auth import authenticate

class CardRetrieve(generics.RetrieveAPIView):

queryset = Card.objects.all()

serializer\_class = CardSerializer

permission\_classes = [permissions.IsAuthenticated]

class DeckList(generics.ListAPIView):

queryset = Deck.objects.all()

serializer\_class = DeckSerializer

class DeckRetrieve(generics.RetrieveAPIView):

queryset = Deck.objects.all()

serializer\_class = DeckSerializer

permission\_classes = [permissions.IsAuthenticated]

def get(self, request, \*args, \*\*kwargs):

deck = self.get\_object()

queryset = (Card.objects.filter(deck = deck)

.exclude(id\_\_in = [progress.card.id for progress in

Progress.objects.filter(user = request.user)]))

serializer = CardSerializer(queryset, many = True)

return Response(serializer.data)

class UserProgressListCreate(generics.ListCreateAPIView):

serializer\_class = UserProgressSerializer

permission\_classes = (permissions.IsAuthenticated,)

def get\_queryset(self):

if self.request.user.is\_staff:

return Progress.objects.all()

return Progress.objects.filter(user = self.request.user.id)

def post(self, request, \*args, \*\*kwargs):

progress\_record = Progress.objects.create(

user = request.user,

card = Card.objects.get(id = request.data.get('card')),

attempts = 0,

successful\_attempts = 0

)

progress\_record.save()

return Response("Card added to your deck successfully!",200)

class UserAnswerCheck(generics.UpdateAPIView):

serializer\_class = UserProgressSerializer

def patch(self, request, action, \*args, \*\*kwargs):

if action == "right":

progress = Progress.objects.get(card = request.data.get('card\_id'), user = request.user)

progress.attempts += 1

progress.successful\_attempts += 1

serializer = self.get\_serializer(progress, data=request.data, partial= True)

serializer.is\_valid(raise\_exception=True)

self.perform\_update(serializer)

return Response(serializer.data)

elif action == "wrong":

progress = Progress.objects.get(card = request.data.get('card\_id'), user = request.user)

progress.attempts += 1

serializer = self.get\_serializer(progress, data=request.data, partial=True)

serializer.is\_valid(raise\_exception=True)

self.perform\_update(serializer)

return Response(serializer.data)

class UserList(generics.ListAPIView):

serializer\_class = UserSerializer

permission\_classes = (permissions.IsAuthenticated,)

def get\_queryset(self):

return get\_user\_model().objects.filter(id = self.request.user.id)

class UserDelete(views.APIView):

def delete(self, request, \*args, \*\*kwargs):

user = request.user

if not user.is\_authenticated:

return Response("Not allowed to do this. You need to login first.", 401)

user.delete()

response = Response({'message': 'Account deleted successfully!'}, 200)

response.delete\_cookie('access\_token')

response.delete\_cookie('refresh\_token')

return response

class UserAuth(views.APIView):

def post(self, request, action, \*args, \*\*kwargs):

if action == "change-password":

if not request.user.is\_authenticated:

return Response("Not allowed to do this. You need to login first.", 401)

user = request.user

serializer = ChangePasswordSerializer(data=request.data, context={'user': user})

if serializer.is\_valid():

new\_password = serializer.validated\_data['new\_password']

user.set\_password(new\_password)

user.save()

return Response({"message": "Password changed successfully."}, status=200)

return Response(serializer.errors, status=400)

elif action == "logout":

if not request.user.is\_authenticated:

return Response("Not allowed to do this. You need to login first.", 401)

refresh\_token = request.COOKIES.get('refresh\_token')

if refresh\_token:

token = RefreshToken(refresh\_token)

token.blacklist()

response = Response({'message': 'Logout successful'})

response.delete\_cookie('refresh\_token')

response.delete\_cookie('access\_token')

return response

elif action == "login":

username = request.data.get('username')

password = request.data.get('password')

user = authenticate(username=username, password=password)

if user is None:

return Response({'error': 'Invalid credentials. Try again!'}, status=401)

refresh = RefreshToken.for\_user(user)

response = Response()

response.set\_cookie(

key='refresh\_token',

value=str(refresh),

secure=False,

httponly=True,

samesite='Lax'

)

response.set\_cookie(

key='access\_token',

value=str(refresh.access\_token),

secure=False,

httponly=True,

samesite='Lax'

)

response.data = {

'message': 'Login successful'

}

return response

elif action == "register":

if get\_user\_model().objects.filter(username = request.data.get('username')).exists():

return Response("User already exists!", 400)

user = get\_user\_model().objects.create(

username = request.data.get('username'),

email = request.data.get('email'),

first\_name = request.data.get('first\_name'),

last\_name = request.data.get('last\_name')

)

user.set\_password(request.data.get('password'))

user.save()

login\_res = self.post(request, action='login')

return login\_res

elif action == 'edit':

if get\_user\_model().objects.filter(username = request.data.get('username')).exists():

return Response("User already exists!", 400)

user = request.user

user.username=request.data.get('username')

user.email=request.data.get('email')

user.first\_name=request.data.get('first\_name')

user.last\_name=request.data.get('last\_name')

user.save()

return Response("Profile editing completed successfully!",200)

else:

return Response(f"Bad Request. We dont have {action}/ page...", status = 400)

Код файла “urls.py”

from django.urls import path

from api.views import UserProgressListCreate, UserAnswerCheck, CardRetrieve, DeckList

from api.views import DeckRetrieve, UserList, UserAuth, UserDelete

urlpatterns = [

path('card/<int:pk>/', CardRetrieve.as\_view()),

path('deck/', DeckList.as\_view()),

path('deck/<int:pk>/', DeckRetrieve.as\_view()),

path('auth/<str:action>/', UserAuth.as\_view()),

path('profile/', UserList.as\_view()),

path('progress/', UserProgressListCreate.as\_view()),

path('progress/<str:action>/', UserAnswerCheck.as\_view()),

path('delete/', UserDelete.as\_view()),

]

Код файла “middleware.py”

from rest\_framework\_simplejwt.exceptions import InvalidToken, TokenError

from rest\_framework\_simplejwt.tokens import RefreshToken

from rest\_framework\_simplejwt.authentication import JWTAuthentication

class RefreshAccessTokenMiddleware:

def \_\_init\_\_(self, get\_response):

self.get\_response = get\_response

def \_\_call\_\_(self, request):

access\_token = request.COOKIES.get('access\_token')

if access\_token:

try:

request.META['HTTP\_AUTHORIZATION'] = f'Bearer {access\_token}'

jwt\_auth = JWTAuthentication()

jwt\_auth.get\_validated\_token(access\_token)

except (InvalidToken, TokenError):

refresh\_token = request.COOKIES.get('refresh\_token')

if refresh\_token:

try:

new\_access\_token = RefreshToken(refresh\_token).access\_token

request.META['HTTP\_AUTHORIZATION'] = f'Bearer {new\_access\_token}'

response = self.get\_response(request)

response.set\_cookie(

key='access\_token',

value=str(new\_access\_token),

httponly=True,

samesite='Lax'

)

response.status\_code = 200

return response

except (InvalidToken, TokenError):

response.data = {'error': 'Invalid refresh token. You need to login again!'}

response.status\_code = 401

response.delete\_cookie('refresh\_token')

response.delete\_cookie('access\_token')

return response

response = self.get\_response(request)

return response

Dockerfile клиентской части

FROM node:20 as build

WORKDIR /app

COPY package\*.json ./

RUN npm ci

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/build /usr/share/nginx/html

COPY --from=build /app/build/static /usr/share/nginx/html/static

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

Dockerfile серверной части

FROM python:3.10

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["gunicorn", "--bind", "0.0.0.0:8000", "cardiogram.wsgi:application"]

Маршрутизация в App.jsx

import React from 'react';

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

import Home from './pages/Home';

import Decks from './pages/Decks';

import Cards from './pages/Cards';

import Learn from './pages/Learn';

import Profile from './pages/Profile';

import MyCards from './pages/MyCards';

import Login from './pages/Login';

import Signup from './pages/Signup';

import ChangePassword from './pages/ChangePassword';

import EditProfile from './pages/EditProfile';

import axios from 'axios';

const App = () => {

axios.defaults.withCredentials = true;

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/decks" element={<Decks />} />

<Route path="/deck/:deckId" element={<Cards />} />

<Route path="/learn" element={<Learn />} />

<Route path="/profile" element={<Profile />} />

<Route path="/my-cards" element={<MyCards />} />

<Route path="/login" element={<Login />} />

<Route path="/signup" element={<Signup />} />

<Route path="/change-password" element={<ChangePassword />} />

<Route path="/edit" element={<EditProfile />} />

</Routes>

</Router>

);

}

export default App;

React компонент страницы обучения Learn.jsx

import React, { useState, useEffect } from 'react';

import Header from '../components/Header';

import CardBack from '../components/CardBack';

import LeftRightButtons from '../components/LeftRightButtons';

import bg from '../assets/static/cards-background.png';

import axios from 'axios';

import styles from '../assets/static/card.module.css';

const Learn = () => {

const [cards, setCards] = useState([]);

const [card, setCard] = useState(null);

const [currentIndex, setCurrentIndex] = useState(0);

const [showBack, setShowBack] = useState(false);

useEffect(() => {

axios

.get('http://127.0.0.1:8000/api/progress/', { withCredentials: true })

.then((res) => {

setCards(res.data);

})

.catch((err) => {

console.error(err);

});

}, []);

useEffect(() => {

if (cards.length && currentIndex !== -1) {

axios

.get(`http://127.0.0.1:8000/api/card/${cards[currentIndex].card}/`, { withCredentials: true })

.then((res) => {

setCard(res.data);

setShowBack(false);

})

.catch((err) => {

console.error(err);

});

}

}, [cards, currentIndex]);

const handleLeft = () => {

if (currentIndex !== -1) {

axios

.patch('http://127.0.0.1:8000/api/progress/right/', { card\_id: cards[currentIndex].card }, { withCredentials: true })

// .then(() => {})

.catch((err) => {

console.error(err);

});

}

if (currentIndex < cards.length - 1 && currentIndex !== -1) {

setCurrentIndex(currentIndex + 1);

} else {

setCurrentIndex(-1);

}

};

const handleRight = () => {

if (currentIndex !== -1) {

axios

.patch('http://127.0.0.1:8000/api/progress/wrong/', { card\_id: cards[currentIndex].card }, { withCredentials: true })

.then(() => {})

.catch((err) => {

console.error(err);

});

}

if (currentIndex < cards.length - 1 && currentIndex !== -1) {

setCurrentIndex(currentIndex + 1);

} else {

setCurrentIndex(-1);

}

};

return (

<div className={styles.cardPage} style={{ backgroundImage: `url(${bg})` }}>

<Header />

<LeftRightButtons

leftFunc={showBack ? handleLeft: () => {setShowBack(true)}}

rightFunc={showBack ? handleRight: () => {setShowBack(true)}}

/>

{currentIndex !== -1 ? (

<CardBack

title={card?.front\_text}

translation={card?.back\_text}

description={card?.example\_usage}

showBack={showBack}

isSmall = {false}

deckId = {card?.deck}

/>

) : (

<p>Карточки закончились! Можете набрать новых или отдохнуть)</p>

)}

</div>

);

};

export default Learn;

Файл docker-compose.yml

services:

db:

image: postgres:latest

container\_name: cardiogram-database

restart: always

volumes:

- postgres\_data:/var/lib/postgresql/data

env\_file:

- .env

backend:

build: cardiogram/

container\_name: cardiogram-backend

ports:

- "8000:8000"

depends\_on:

- db

restart: always

env\_file:

- .env

frontend:

build: cardiogram-front/

container\_name: cardiogram-frontend

ports:

- "80:80"

depends\_on:

- backend

restart: always

volumes:

postgres\_data: