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 контенеризованы 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.

Код docker-compose файла приведен в приложении Б.

1. Описание API

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,  fisrt\_name,  last\_name,  email,  password | {“username”: “Werd”,  “fisrt\_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,  fisrt\_name,  last\_name,  email | {“username” : “Werd”,  “fisrt\_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”,  “fisrt\_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.

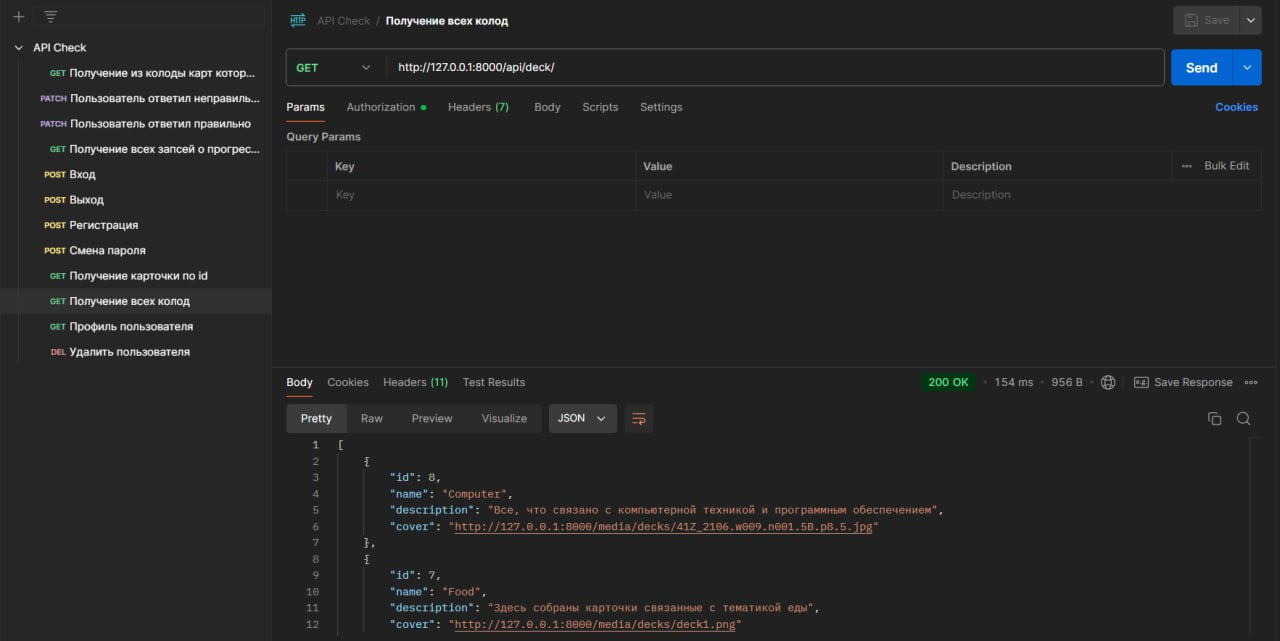


Рисунок 1 – Скриншот успешного запроса информации о колодах

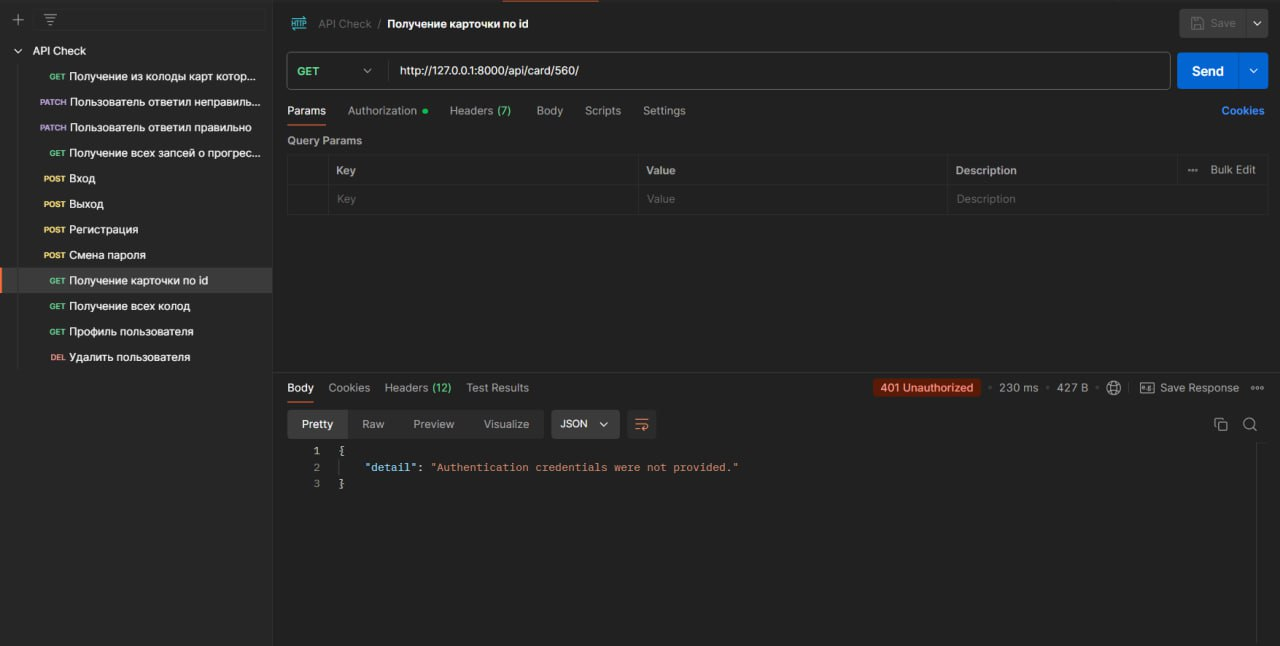


Рисунок 2 – Ошибка: пользователь не прошел аутентификацию

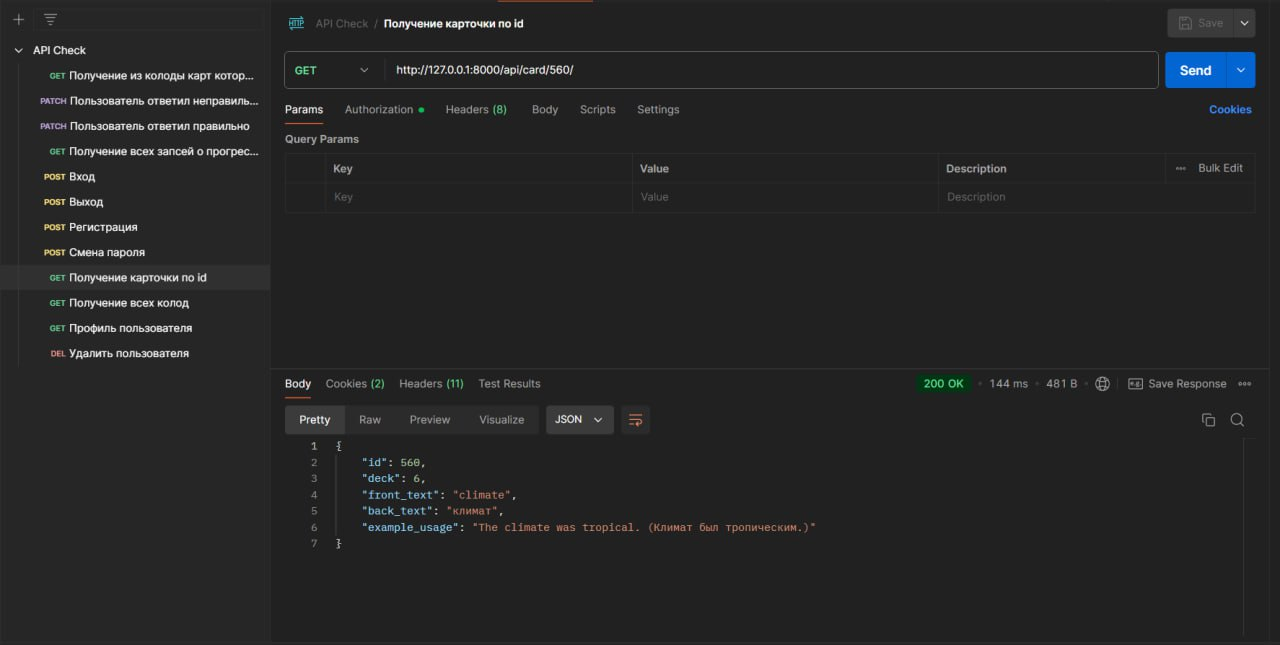


Рисунок 3 – Успешный запрос информации о карточке

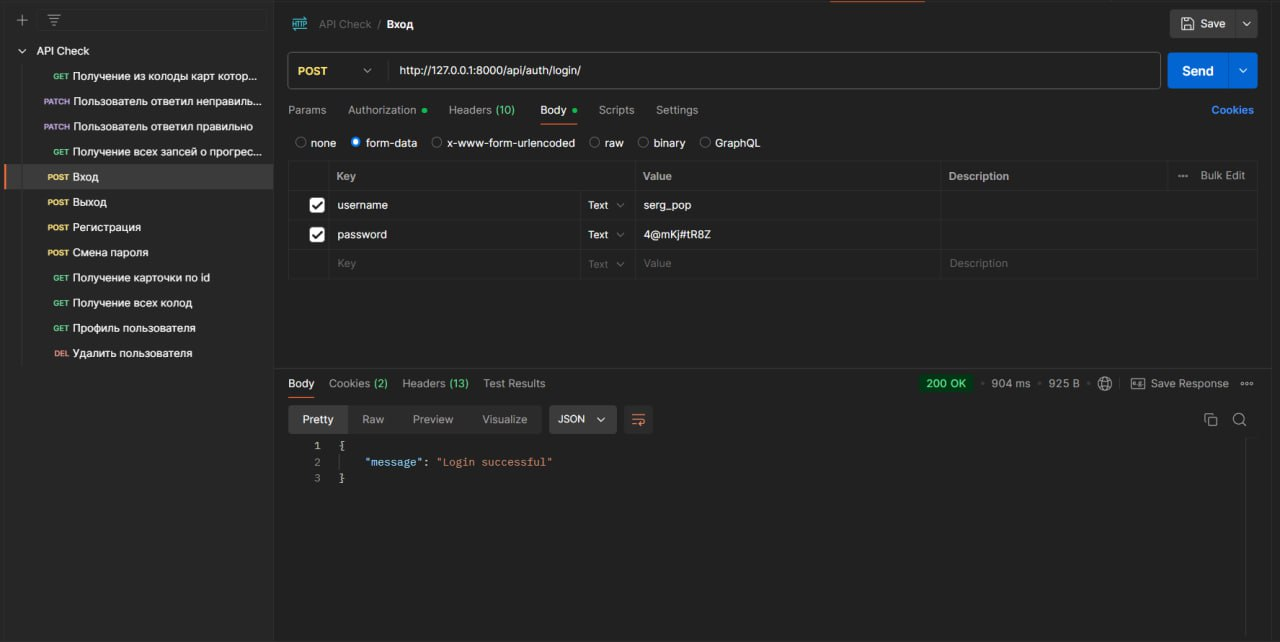


Рисунок 4 – Успешный запрос на вход в систему

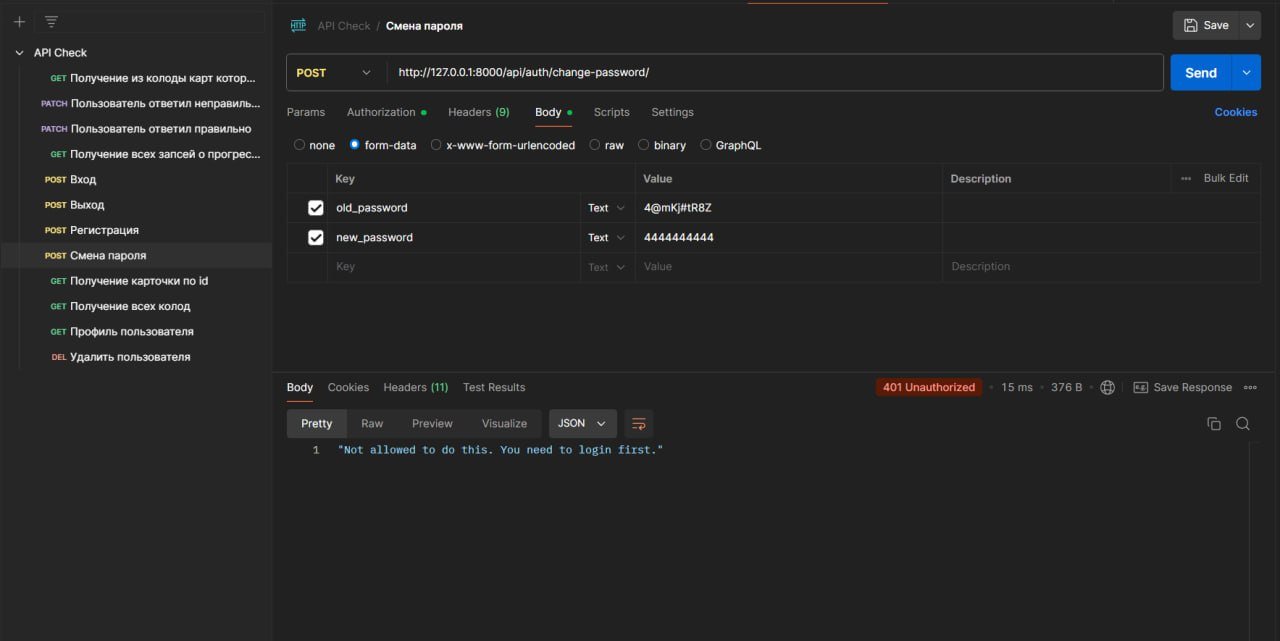


Рисунок 5 – Ошибка: не аутентифицированный пользователь не может менять пароль

1. Аутентификация и авторизация

Серверное приложение использует механизм JTW-аутентификации для управления доступом пользователей. В качестве механизма авторизации выступает разрешение isAuthenticated позволяющее доступ к определенным эндпоинтам только аутентифицированным пользователям.

* 1. JWT-аутентификация

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.

1. Пользовательский интерфейс

Пользовательский интерфейс приложения реализован с помощью JavaScript библиотеки React и 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 в темами колод иностранных слов;
* Карты в колоде, карты пользователя, страница обучения – доступны авторизованным пользователям, позволяют увидеть все карты представленные в приложении.

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.

1. Структура контейнеризации и настройки окружения

Все компоненты конечной системы контенеризованы с помощью Docker для обеспечения изоляции окружения, удобства развертывания и масштабирования.

* 1. Архитектура контейнеров

Архитектура контейнеров включает в себя 3 компонента:

* 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 собранное React приложение с настроенное для 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 конфигурацию всех компонентов и их связи.

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.

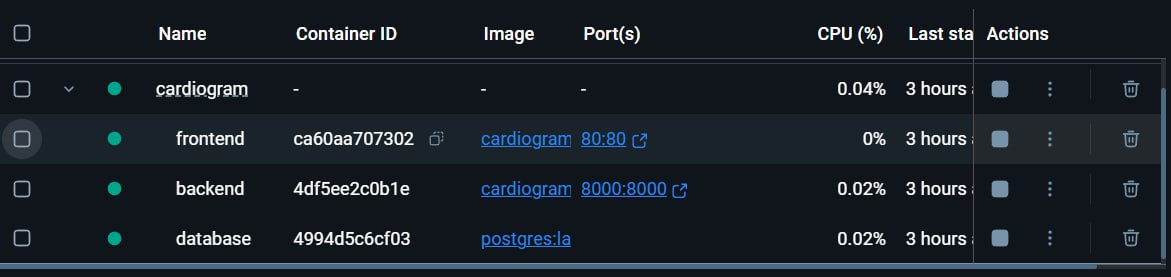


Рисунок 6 – Запущенные контейнеры в приложении Docker Desktop

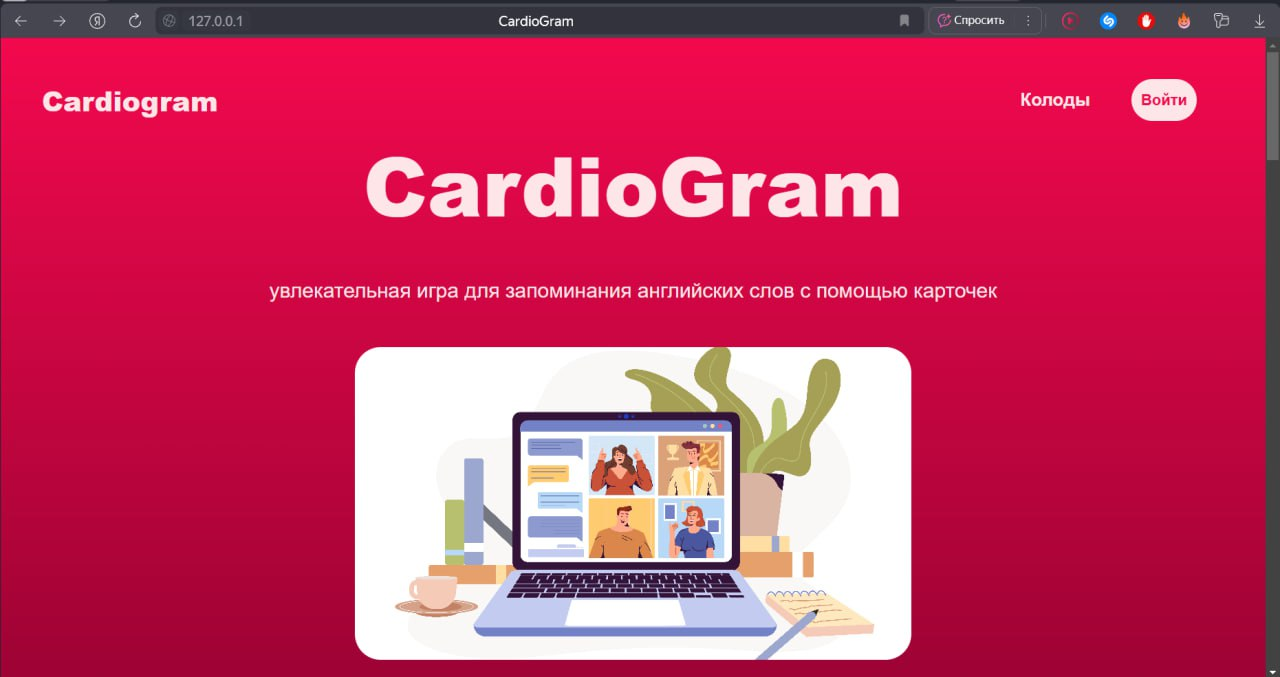


Рисунок 7 – Главная страница приложения в браузере

Приложение А

Ссылка на GitHub с проектом

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

Приложение Б

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:

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;