МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение   
высшего образования «Самарский национальный исследовательский университет   
имени академика С.П. Королева»

(Самарский университет)

Институт информатики и кибернетики

Кафедра технической кибернетики

**ОТЧЕТ ПО ЛАБОРАТОРНЫМ РАБОТАМ**

Дисциплина: «Технологии сетевого программирования»

Выполнили студенты

группы 6301-010302D

Астафьева А.В.

и 6304-010302D

Серебряков В.С.

САМАРА 2025

реферат

**Отчет по лабораторным работам:** 75с., 12 рисунков, 2 приложения.

веб-приложение, клиент-серверная архитектура, rest api, python, javascript, docker, сетевое программирование, интерфейс пользователя, аутентификация

**Отчет по лабораторным работам** по дисциплине «Технологии сетевого программирования» посвящен разработке веб-приложения **WeatherToday**, предназначенного для просмотра текущей погоды в различных городах.

Приложение построено по **клиент-серверной архитектуре** и включает три основных компонента:

* **Backend** (Python, Flask) – отвечает за обработку запросов, взаимодействие с внешним API openweathermap.org и работу с базой данных.
* **Frontend** (JavaScript, React) – обеспечивает пользовательский интерфейс с навигационной панелью, карточками погоды и профилем пользователя.
* **База данных** (PostgreSQL) – хранит информацию о пользователях, городах, погодных данных и избранных локациях.

Для удобства развертывания и масштабируемости приложение **контейнеризировано** с использованием **Docker** и **Docker Compose**. В ходе работы были реализованы **12 API-эндпоинтов**, обеспечивающих регистрацию пользователей, управление избранными городами, получение и обновление погодных данных.

Тестирование API проводилось в **Postman**, а интерфейс приложения включает **главную страницу с погодой**, **профиль пользователя** и **формы авторизации**.

Разработка проекта позволила закрепить навыки работы с **сетевыми технологиями**, **REST API**, **реляционными базами данных** и **контейнеризацией**.

содержание

[Введение 5](#_Toc199765597)

[1 Архитектура 7](#_Toc199765598)

[2 Описание API 8](#_Toc199765599)

[3 Описание базы данных 10](#_Toc199765600)

[4 Тестирование API 12](#_Toc199765601)

[5 Интерфейс 15](#_Toc199765602)

[6 Контейнеризация приложения 18](#_Toc199765603)

[Заключение 21](#_Toc199765604)

[Приложение A 22](#_Toc199765605)

[Приложение B 23](#_Toc199765606)

ВВЕДЕНИЕ

В данной работе рассматривается процесс разработки веб-приложения **WeatherToday**, предназначенного для отображения текущей погоды в городах по выбору пользователя.

**Актуальность проекта** обусловлена растущим спросом на удобные и функциональные сервисы прогнозирования погоды, а также необходимостью освоения современных технологий сетевого программирования, включая **REST API, контейнеризацию и работу с базами данных**.

**Цель работы** – создание полноценного клиент-серверного приложения с использованием **Python (Flask)** для backend-части, **JavaScript (React)** для frontend-части и **PostgreSQL** в качестве системы управления базами данных. Дополнительной задачей является интеграция с внешним API **OpenWeatherMap** для получения актуальных метеорологических данных.

**Основные задачи проекта:**

1. Разработка **серверной части (backend)** на Flask с реализацией RESTful API.
2. Создание **клиентского интерфейса (frontend)** на React для удобного взаимодействия с пользователем.
3. Проектирование и настройка **базы данных** для хранения информации о пользователях, городах и погодных данных.
4. Настройка **Docker-контейнеров** для backend, frontend и PostgreSQL с использованием **Docker Compose** для упрощения развертывания.
5. Тестирование API с помощью **Postman** и проверка корректности работы всех функций приложения.

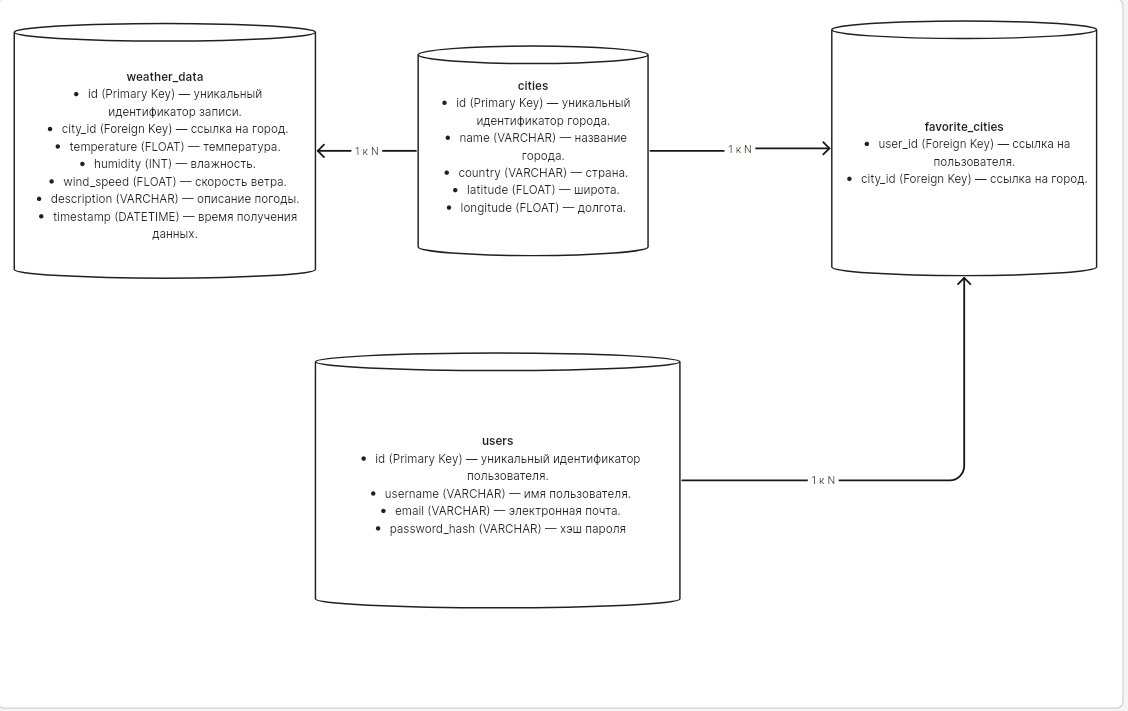
**Методология разработки** включает применение **микросервисной архитектуры**, что позволяет разделить функционал приложения на независимые компоненты (backend, frontend, БД), упрощая их масштабирование и поддержку. Для обеспечения безопасности реализована **аутентификация пользователей** через JWT (JSON Web Tokens).

**Практическая значимость работы** заключается в демонстрации полного цикла разработки веб-приложения – от проектирования API и интерфейса до развертывания в контейнерах. Полученные навыки могут быть применены в создании аналогичных сервисов, таких как системы мониторинга данных, геосервисы или чат-боты с интеграцией внешних API.

1. Архитектура

Для написания приложения использовалась архитектура клиент-сервер. Оно состоит из 3-х компонентов, находящихся в разных контейнерах и связанных с помощью Docker Compose. Основные части:

* Backend (Python и его библиотеки) базировался на микрофреймворке Flask.
* Frontend (JavaScript + CSS) написан с помощью библиотеки React.
* База данных (PostgreSQL). Описание базы данных:

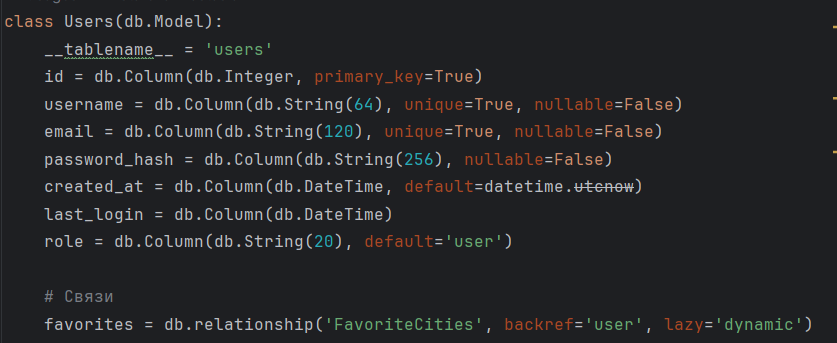


1. Описание API

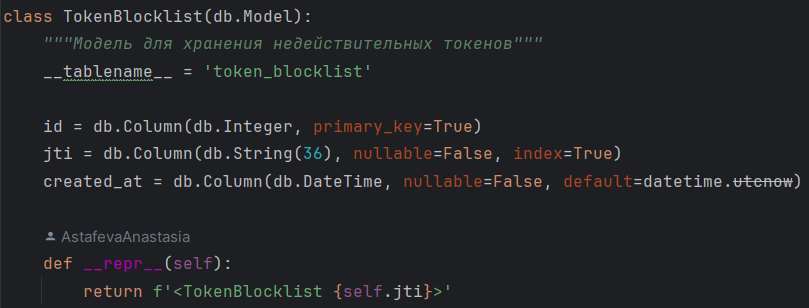
Для связи клиентской и серверной части, а также для связи приложения с внешним сервисом openweathermap.org были написаны следующие API ручки:

1. delete\_user(user\_id) – удаление пользователя бд по id. URL: /users/<int:user\_id>. Метод – DELETE.
2. get\_user(user\_id) – получение данных аккаунта для пользователя. URL: /users/<int:user\_id>. Метод – GET.
3. get\_favorite\_cities\_weather(user\_id) –получение текущей погоды для любимых городов пользователя. URL: /users/<int:user\_id>/favorites/weather. Метод – GET.
4. search\_user() – поиск пользователей по имени или email. URL: /users/search. Метод – GET.
5. add\_city() – добавление в базу данных города по названию из openweathermap.org. URL: /cities. Метод – POST.
6. delete\_city() – удаление города из БД по id. URL: /cities/<int:city\_id>. Метод – DELETE.
7. add\_favorite\_city(user\_id) – добавление любимого города в свой аккаунт. URL: /users/<int:user\_id>/favorities. Метод – POST.
8. remove\_favorite\_city(user\_id, city\_id) – удаление города из любимых. URL: /users/<int:user\_id>/favorities/<int:city\_id>. Метод – DELETE.
9. get\_city\_weather – получение всех записей о погоде для города по id. URL: /weather/<int:city\_id>. Метод – GET.
10. update\_hourly\_weather(city\_id) – добавляет почасовые данные о погоде для города по id. URL: /weather/update\_hourly/<int:city\_id>. Метод – POST.
11. cleanup\_old\_weather\_data() – удаляет устаревшие записи о погоде. URL: /weather/cleanup. Метод – DELETE.
12. register() – регистрация пользователя. URL: /register. Метод – POST.
13. refresh() – метод для обновления acsess токена. URL: /refresh. Метод – POST.
14. update\_account() – изменение пароля или email. URL: /update-account. Метод – POST.
15. Описание базы данных

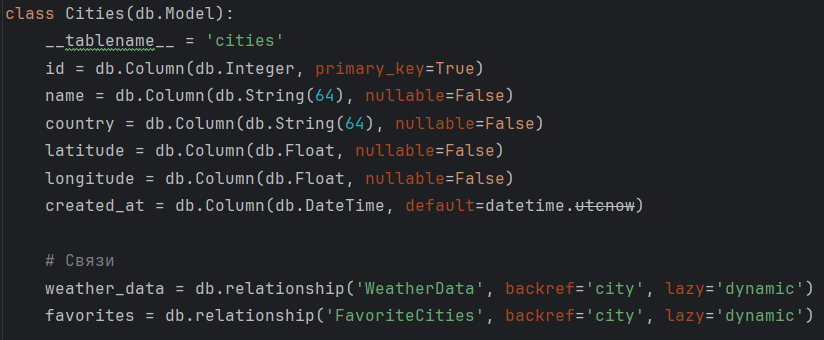
Модель Users:



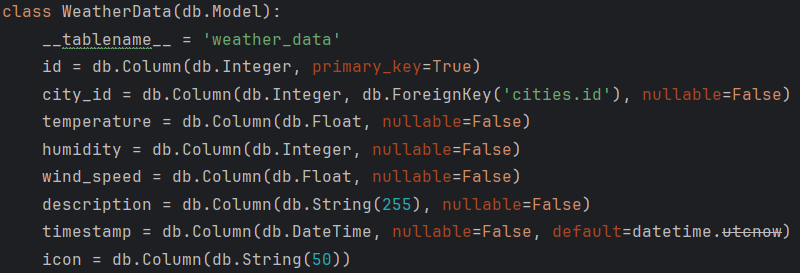
Модель для хранения недействительных токенов:



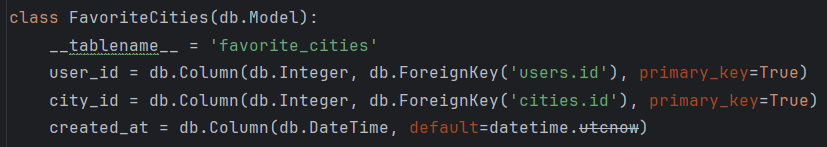
Модель городов:



Модель погоды:



Модель любимых городов:



1. Тестирование API

Чтобы протестировать работу API использовался Postman.

Примеры запросов и ответов:

Пример корректного запроса по эндпоинту /login изображен на рисунке 1.

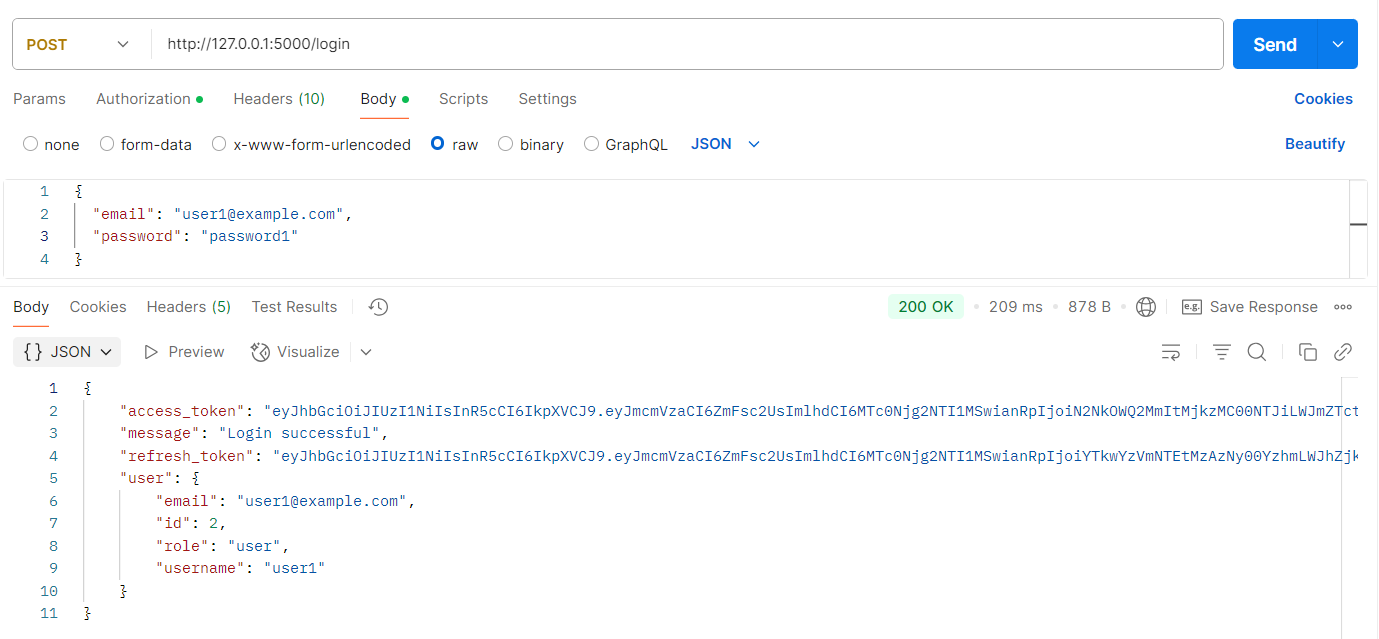


Рисунок 1 – пример корректного запроса входа пользователя

На рисунке 2 изображен неверный запрос по эндпоинту /login (неверный пароль).

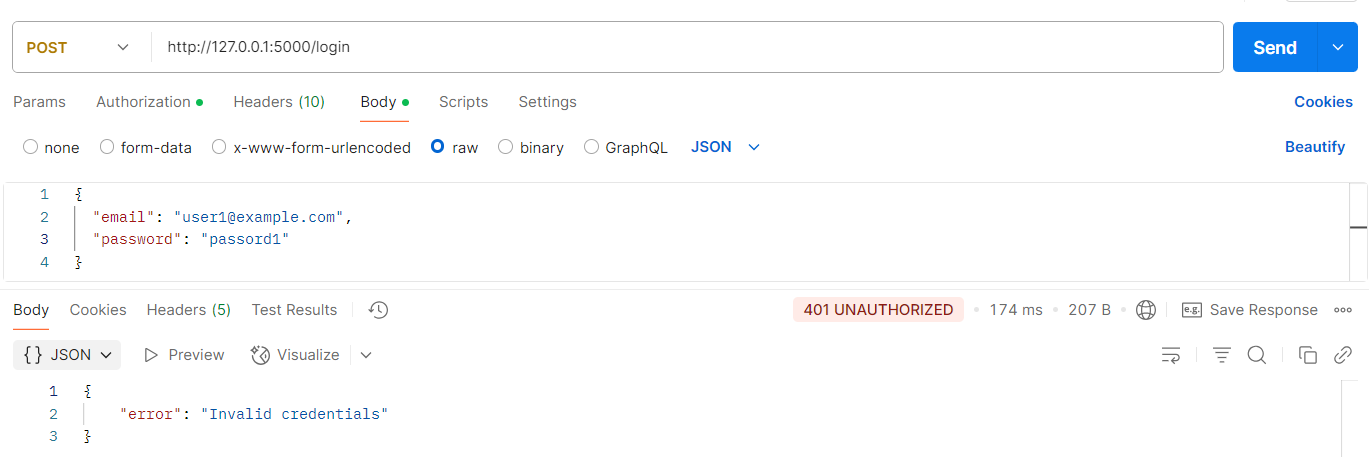


Рисунок 2 – неправильный запрос для входа пользователя

На рисунке 3 изображен успешный запрос на выход из аккаунта пользователя.

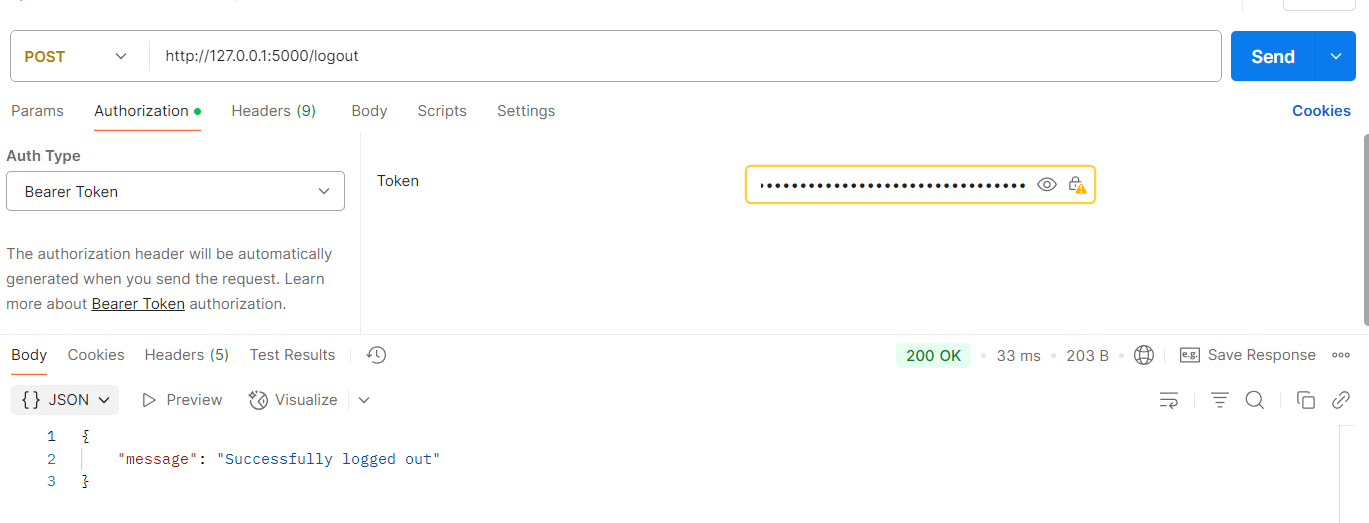


Рисунок 3 – успешный запрос на выход

На рисунке 4 изображен некорректный запрос по эндпоинту /logout (неверный acsess-токен).

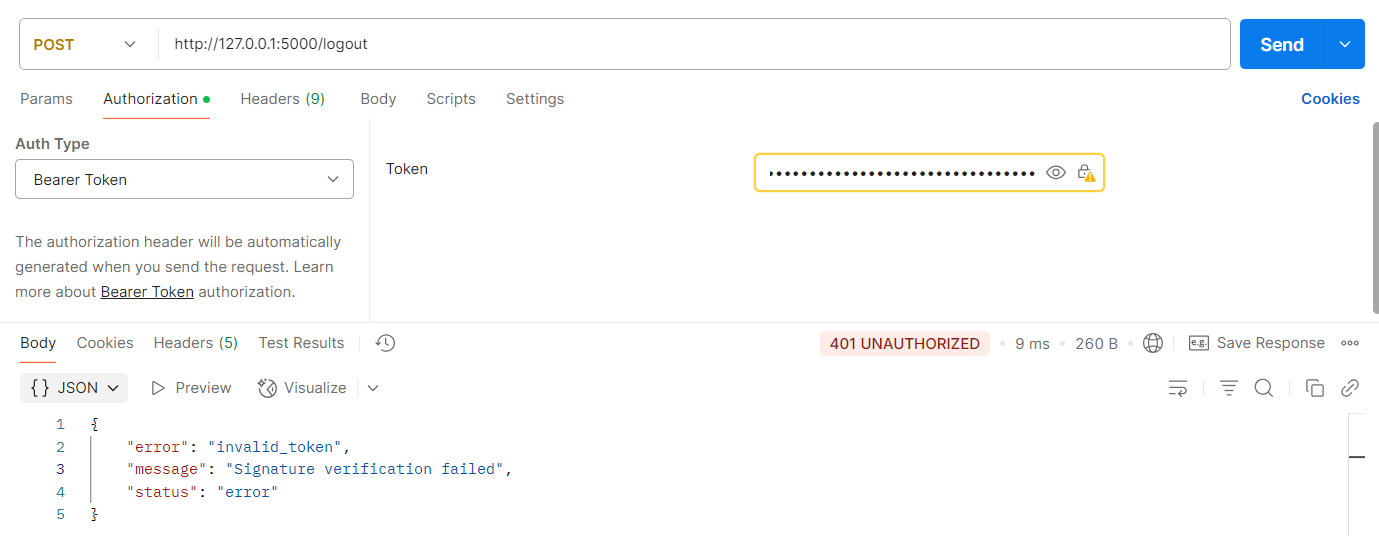


Рисунок 4 – некорректный запрос на выход

На рисунке 5 изображен успешный запрос по эндпоинту /register.

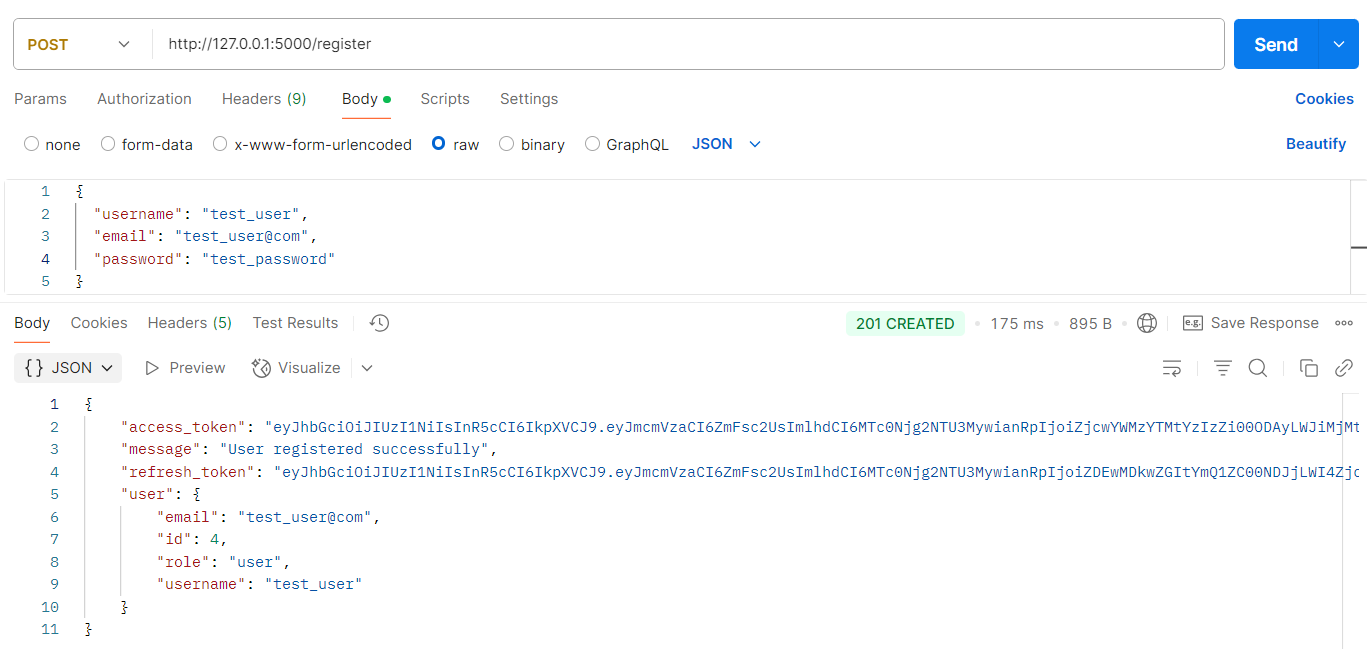


Рисунок 5 – успешный запрос на регистрацию

1. Интерфейс

Интерфейс состоит из 2-х основных частей, изображенных на рисунке 6: Страница и Навигационная панель.

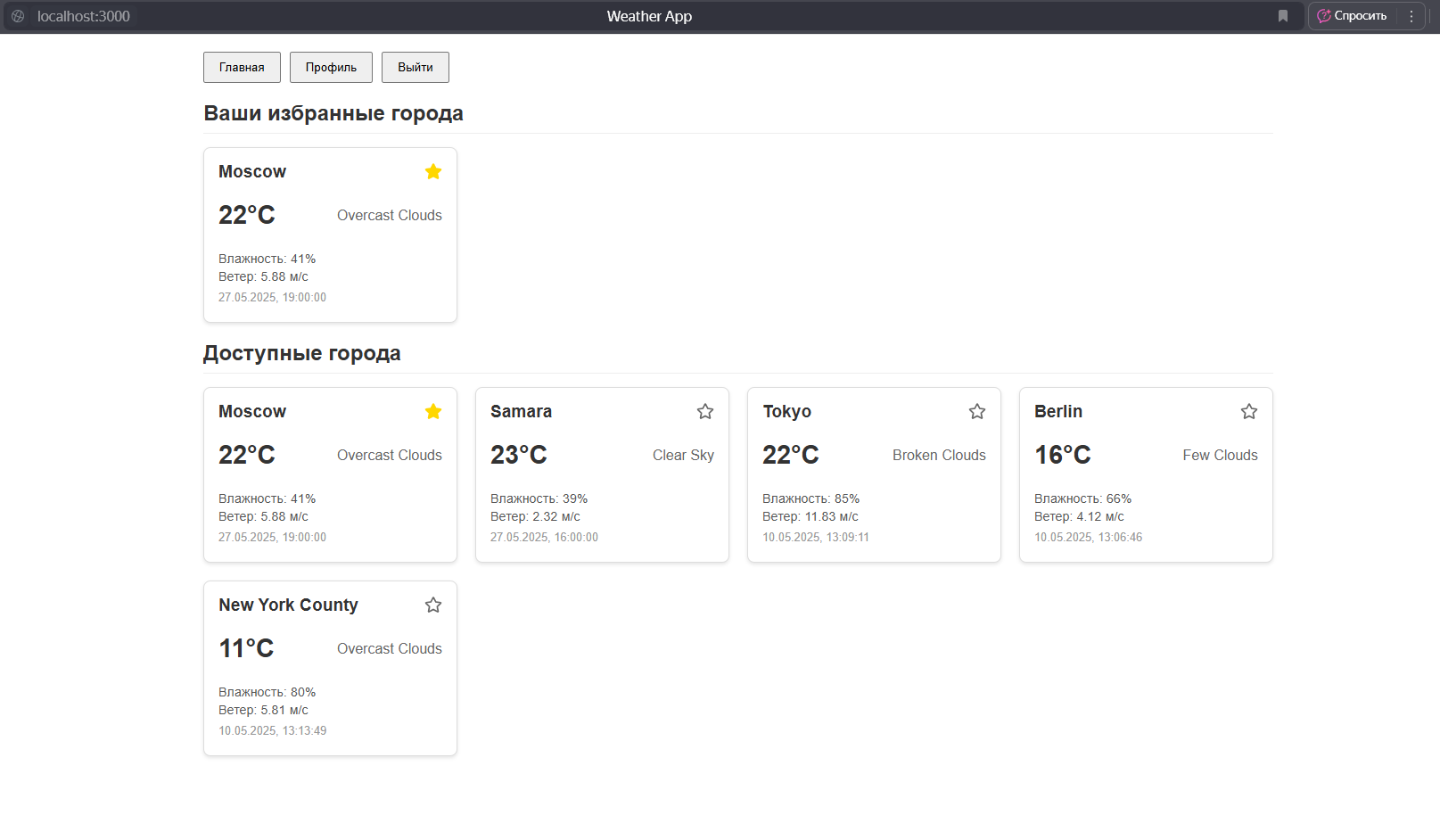


Рисунок 6 – основные части интерфейса

В Навигационной панели (верхняя часть экрана с кнопками) размещены кнопки для перехода по страницам (Главная, профиль, выход).

На главной странице размещена информация по погоде. На этой же странице можно "закрепить" город в избранном. Функция закрепления доступна только зарегистрированным пользователям.

Кнопка "профиль" перенаправляет пользователя на страницу, на которой он сможет увидеть информацию о своем профиле и изменить информацию о себе (сменить логин, почту, пароль). Эта страница изображена на рисунке 7.

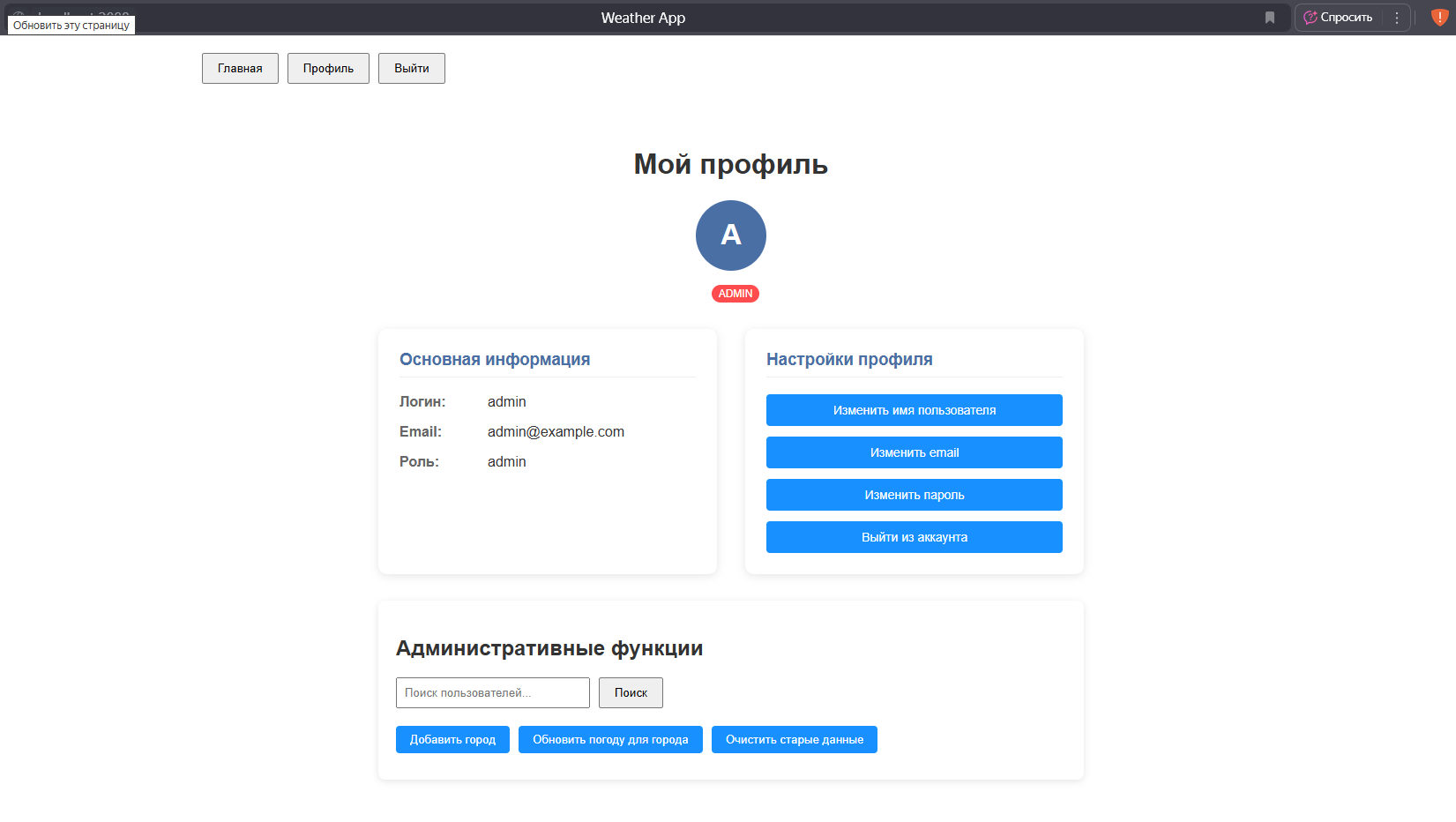


Рисунок 7 – страница профиля

Если пользователь не вошел в систему, вместо кнопки выйти будет кнопка входа, изображенная на рисунке 8, которая перенаправит пользователя на форму входа или регистрации.

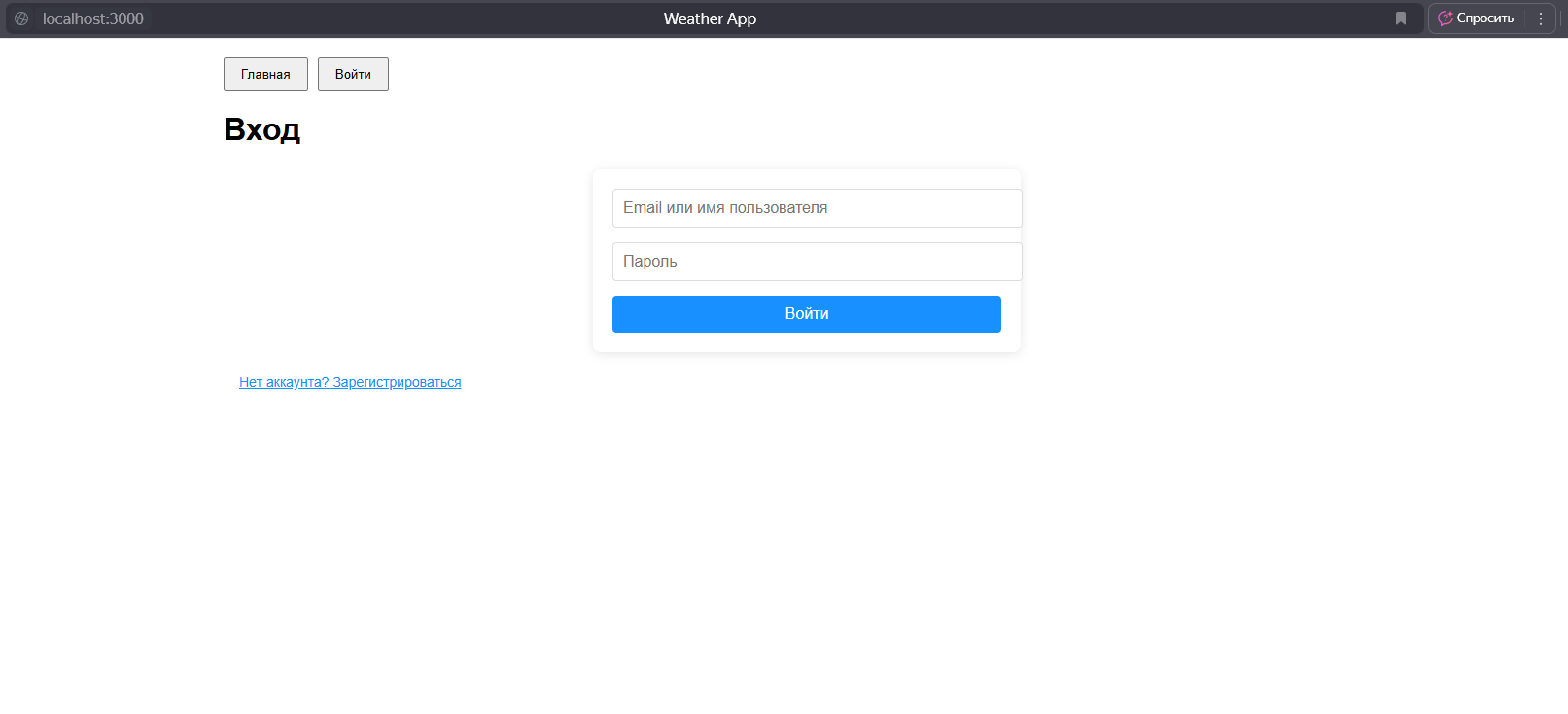


Рисунок 8 – страница входа

Страница регистрации изображена на рисунке 9.

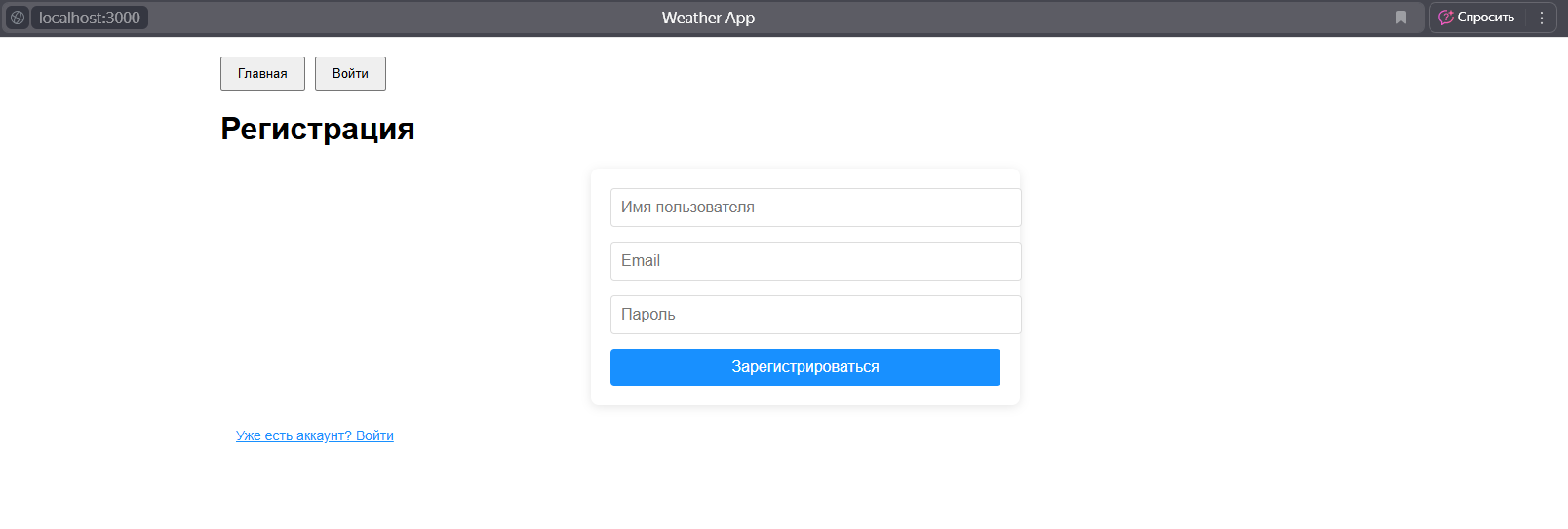


Рисунок 9 – страница регистрации

Кнопка выход – выход из системы.

Структура:

* api/ – содержит API для связи с сервером,
* components/ – содержит такие компоненты, как Authform.js – форма регистрации/авторизации, Navbar.js – навигационная панель для переключения по страницам, Weathercard.js – функционал карточек погоды. В каждой карточке по api получаются данные по погоде и городу и выводятся на главной странице,
* pages/ – содержит страницы авторизации (Auth.js), страницу с погодой и избранными городами (Home.js), страинцу с профилем пользователя (Profile.js),
* styles/ – папка с файлами стилей страниц формата .css,
* App.js – файл приложения.

1. Контейнеризация приложения

Для упаковки и развертывания приложения был использован Docker, Docker Compose и базу данных Postgres. Это позволяет легче запускать приложение на любых серверах.

Dockerfile для бэкэнда изображен на рисунке 10.

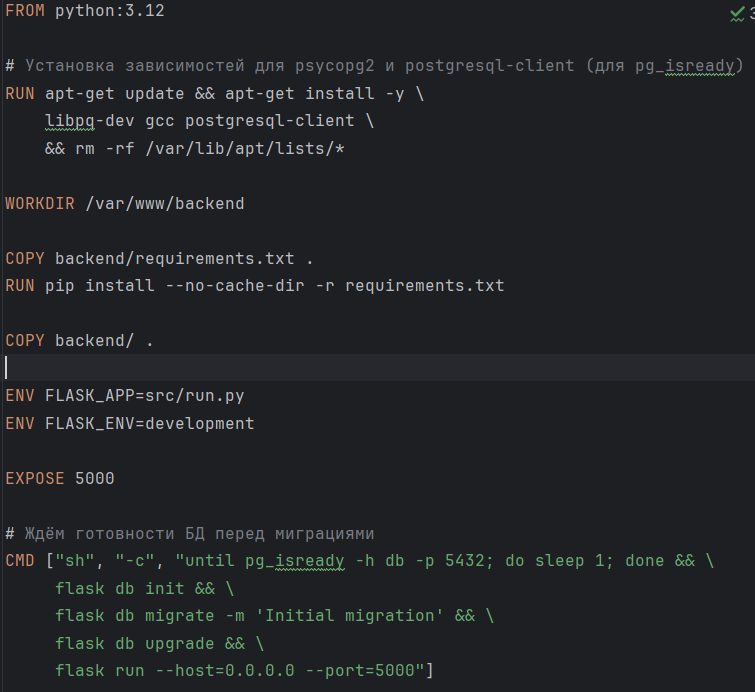
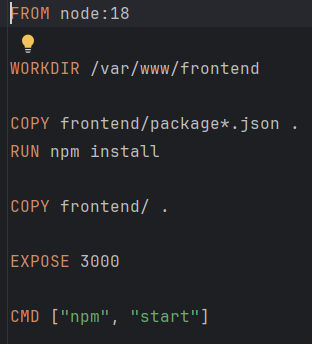


Рисунок 10 – Dockerfile для бэкэнда

Он построен на основе Python 3.12 и выполняет сборку Flask-приложения. Установка зависимостей для psycopg2 и postgresql-client нужна для корректной настройки миграции при запуске бэкэнда. Рабочая директория – /var/www/backend. Копируется файл requirements.txt в корень рабочей директории и устанавливаются все необходимые библиотеки для работы серверной части приложения. Далее идет установка переменных окружения (в первую очередь FLASK\_APP=src/run.py для корректного запуска flask приложения) и открывается порт 5000.

Dockerfile для фронтэнда изображен на рисунке 11.



Он использует node 18. Затем устанавливается рабочая директория /var/www/frontend и из папки проекта копируется package\*.json файлы, в которых указаны все node modules необходимые для работы. Устанавливаются все npm и весь проект копируется из папки frontend. Открывается 3000 порт для прослушивания.

Dockerfile для БД: Dockerfile использует postgres последней версии (postgres:latest). Устанавливаются название бд, имя юзера и пароль, открывается порт 5432.

Работающее приложение в контейнерах изображено на рисунке 12.

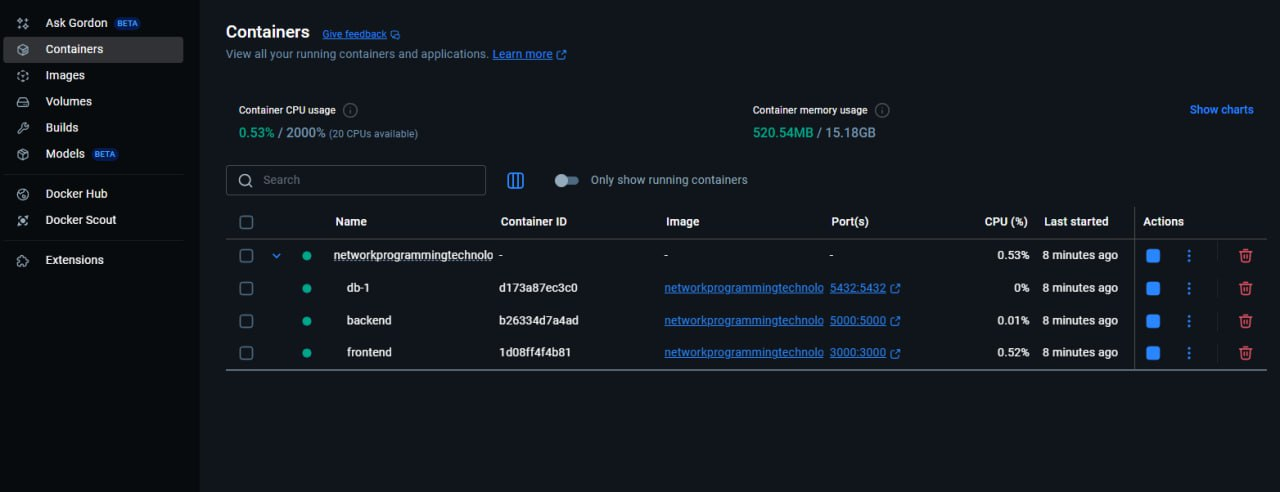


Рисунок 12 – работающее приложение в контейнерах.

заключение

В ходе выполнения лабораторных работ было разработано **веб-приложение WeatherToday**, позволяющее пользователям просматривать актуальную погоду в различных городах, добавлять локации в избранное и управлять своим профилем.

**Основные результаты работы:**

1. Реализована **клиент-серверная архитектура** с использованием **Flask (Python)** для backend и **React (JavaScript)** для frontend.
2. Настроено взаимодействие с внешним API **openweathermap.org** для получения погодных данных.
3. Разработана **база данных PostgreSQL** с таблицами пользователей, городов, погоды и избранных локаций.
4. Созданы **12 API-эндпоинтов**, обеспечивающих полный функционал приложения.
5. Проведено **тестирование API** с помощью Postman, подтвердившее корректность работы всех методов.
6. Реализован **удобный интерфейс** с навигацией, карточками погоды и формой авторизации.
7. Приложение **упаковано в Docker-контейнеры**, что упрощает его развертывание на любом сервере.

**Перспективы развития проекта:**

* Добавление прогноза погоды на несколько дней.
* Интеграция с картографическими сервисами (Google Maps, OpenStreetMap).
* Реализация push-уведомлений о резких изменениях погоды.
* Оптимизация запросов к API для уменьшения нагрузки на сервер.

Разработка данного приложения позволила углубить понимание **сетевых технологий**, **RESTful API**, **работы с базами данных** и **контейнеризации**

ПРИЛОЖЕНИЕ A

GitHub репозиторий проекта: <https://github.com/AstafevaAnastasia/NetworkProgrammingTechnologies>

ПРИЛОЖЕНИЕ B

Код файла run.py:

import sys  
from pathlib import Path  
from flask import Flask, jsonify  
from flask\_cors import CORS  
from flask\_migrate import Migrate  
from flask\_sqlalchemy import SQLAlchemy  
from datetime import datetime, timedelta, timezone  
from functools import wraps  
from flask\_jwt\_extended import (  
 JWTManager,  
 create\_access\_token,  
 create\_refresh\_token,  
 jwt\_required,  
 get\_jwt\_identity,  
 set\_access\_cookies,  
 unset\_jwt\_cookies,  
 verify\_jwt\_in\_request,  
 get\_jwt  
)  
  
sys.path.append(str(Path(\_\_file\_\_).parent.parent.parent))  
  
db = SQLAlchemy()  
migrate = Migrate()  
  
def create\_app():  
 app = Flask(\_\_name\_\_)  
  
 CORS(app,  
 resources={r"/\*": {"origins": "\*"}}, # Или конкретный домен, например "http://localhost:3000"  
 supports\_credentials=True,  
 allow\_headers=["Authorization", "Content-Type"])  
  
 # Конфигурация  
 app.config['SQLALCHEMY\_DATABASE\_URI'] = 'postgresql://postgres:appasonya2@localhost:5432/postgres'  
 app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
  
 # JWT Configuration  
 app.config['JWT\_SECRET\_KEY'] = 'your-extra-super-secret-key-here' # Должен быть надежным!  
 app.config['JWT\_ACCESS\_TOKEN\_EXPIRES'] = timedelta(hours=5)  
 app.config['JWT\_REFRESH\_TOKEN\_EXPIRES'] = timedelta(days=30)  
 app.config['JWT\_TOKEN\_LOCATION'] = ['headers']  
 app.config['JWT\_HEADER\_NAME'] = 'Authorization'  
 app.config['JWT\_HEADER\_TYPE'] = 'Bearer'  
 app.config['JWT\_COOKIE\_SECURE'] = False # True в production  
 app.config['JWT\_COOKIE\_CSRF\_PROTECT'] = False # True в production  
 app.config['JWT\_ACCESS\_COOKIE\_PATH'] = '/'  
 app.config['JWT\_REFRESH\_COOKIE\_PATH'] = '/'  
 app.config['JWT\_BLACKLIST\_ENABLED'] = True  
 app.config['JWT\_BLACKLIST\_TOKEN\_CHECKS'] = ['access', 'refresh']  
  
 jwt = JWTManager(app)  
  
 # Callbacks для обработки ошибок JWT  
 @jwt.expired\_token\_loader  
 def expired\_token\_callback(jwt\_header, jwt\_payload):  
 return jsonify({  
 "status": "error",  
 "message": f"Token expired at {datetime.fromtimestamp(jwt\_payload['exp'])}",  
 "error": "token\_expired",  
 "expired\_at": jwt\_payload['exp']  
 }), 401  
  
 @jwt.invalid\_token\_loader  
 def invalid\_token\_callback(error):  
 if "Token must be a string" in str(error):  
 return jsonify({  
 "status": "error",  
 "message": "Token must be provided as a string",  
 "error": "invalid\_token\_format"  
 }), 401  
 return jsonify({  
 "status": "error",  
 "message": str(error),  
 "error": "invalid\_token"  
 }), 401  
  
 @jwt.unauthorized\_loader  
 def missing\_token\_callback(error):  
 return jsonify({  
 "status": "error",  
 "message": "Authorization required",  
 "error": "authorization\_required"  
 }), 401  
  
 @jwt.needs\_fresh\_token\_loader  
 def token\_not\_fresh\_callback(jwt\_header, jwt\_payload):  
 return jsonify({  
 "status": "error",  
 "message": "Fresh token required",  
 "error": "fresh\_token\_required"  
 }), 401  
  
 @jwt.revoked\_token\_loader  
 def revoked\_token\_callback(jwt\_header, jwt\_payload):  
 return jsonify({  
 "status": "error",  
 "message": "Token has been revoked",  
 "error": "token\_revoked"  
 }), 401  
  
 @jwt.user\_identity\_loader  
 def user\_identity\_lookup(user):  
 # Всегда возвращаем строку!  
 return str(user.id) if hasattr(user, 'id') else str(user)  
  
 @jwt.user\_lookup\_loader  
 def user\_lookup\_callback(\_jwt\_header, jwt\_data):  
 identity = jwt\_data["sub"]  
 # Преобразуем строку в число для поиска в БД  
 return Users.query.get(int(identity))  
  
 # Инициализация расширений  
 db.init\_app(app)  
 migrate.init\_app(app, db)  
  
 # Регистрация Blueprint  
 from backend.src.databases.routes import bp  
 app.register\_blueprint(bp)  
  
 with app.app\_context():  
 from backend.src.databases.models import Users, initialize\_data  
 db.create\_all()  
 if not Users.query.first():  
 initialize\_data()  
 print("Тестовые данные успешно добавлены")  
  
 return app  
  
app = create\_app()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(debug=True)

Код файла models.py:

from backend.src.run import db  
from werkzeug.security import generate\_password\_hash, check\_password\_hash  
from datetime import datetime  
from sqlalchemy.orm import sessionmaker  
from sqlalchemy.exc import IntegrityError  
  
  
class TokenBlocklist(db.Model):  
 *"""Модель для хранения недействительных токенов"""* \_\_tablename\_\_ = 'token\_blocklist'  
  
 id = db.Column(db.Integer, primary\_key=True)  
 jti = db.Column(db.String(36), nullable=False, index=True)  
 created\_at = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)  
  
 def \_\_repr\_\_(self):  
 return f'<TokenBlocklist {self.jti}>'  
  
  
class Users(db.Model):  
 \_\_tablename\_\_ = 'users'  
 id = db.Column(db.Integer, primary\_key=True)  
 username = db.Column(db.String(64), unique=True, nullable=False)  
 email = db.Column(db.String(120), unique=True, nullable=False)  
 password\_hash = db.Column(db.String(256), nullable=False)  
 created\_at = db.Column(db.DateTime, default=datetime.utcnow)  
 last\_login = db.Column(db.DateTime)  
 role = db.Column(db.String(20), default='user')  
  
 # Связи  
 favorites = db.relationship('FavoriteCities', backref='user', lazy='dynamic')  
  
 def \_\_init\_\_(self, \*\*kwargs):  
 super(Users, self).\_\_init\_\_(\*\*kwargs)  
 if 'password' in kwargs:  
 self.set\_password(kwargs['password'])  
  
 def set\_password(self, password):  
 *"""Устанавливает хеш пароля"""* self.password\_hash = generate\_password\_hash(password)  
  
 def check\_password(self, password):  
 *"""Проверяет пароль"""* return check\_password\_hash(self.password\_hash, password)  
  
 def get\_id(self):  
 *"""Возвращает ID пользователя (для Flask-Login)"""* return str(self.id)  
  
 def to\_dict(self):  
 *"""Сериализация пользователя в словарь"""* return {  
 'id': self.id,  
 'username': self.username,  
 'email': self.email,  
 'created\_at': self.created\_at.isoformat() if self.created\_at else None,  
 'last\_login': self.last\_login.isoformat() if self.last\_login else None  
 }  
  
  
class Cities(db.Model):  
 \_\_tablename\_\_ = 'cities'  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String(64), nullable=False)  
 country = db.Column(db.String(64), nullable=False)  
 latitude = db.Column(db.Float, nullable=False)  
 longitude = db.Column(db.Float, nullable=False)  
 created\_at = db.Column(db.DateTime, default=datetime.utcnow)  
  
 # Связи  
 weather\_data = db.relationship('WeatherData', backref='city', lazy='dynamic')  
 favorites = db.relationship('FavoriteCities', backref='city', lazy='dynamic')  
  
 def to\_dict(self):  
 *"""Сериализация города в словарь"""* return {  
 'id': self.id,  
 'name': self.name,  
 'country': self.country,  
 'coordinates': {  
 'latitude': self.latitude,  
 'longitude': self.longitude  
 },  
 'created\_at': self.created\_at.isoformat() if self.created\_at else None  
 }  
  
  
class WeatherData(db.Model):  
 \_\_tablename\_\_ = 'weather\_data'  
 id = db.Column(db.Integer, primary\_key=True)  
 city\_id = db.Column(db.Integer, db.ForeignKey('cities.id'), nullable=False)  
 temperature = db.Column(db.Float, nullable=False)  
 humidity = db.Column(db.Integer, nullable=False)  
 wind\_speed = db.Column(db.Float, nullable=False)  
 description = db.Column(db.String(255), nullable=False)  
 timestamp = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)  
 icon = db.Column(db.String(50))  
  
 def to\_dict(self):  
 *"""Сериализация данных о погоде"""* return {  
 'id': self.id,  
 'city\_id': self.city\_id,  
 'temperature': self.temperature,  
 'humidity': self.humidity,  
 'wind\_speed': self.wind\_speed,  
 'description': self.description,  
 'timestamp': self.timestamp.isoformat() if self.timestamp else None,  
 'icon': self.icon  
 }  
  
  
class FavoriteCities(db.Model):  
 \_\_tablename\_\_ = 'favorite\_cities'  
 user\_id = db.Column(db.Integer, db.ForeignKey('users.id'), primary\_key=True)  
 city\_id = db.Column(db.Integer, db.ForeignKey('cities.id'), primary\_key=True)  
 created\_at = db.Column(db.DateTime, default=datetime.utcnow)  
  
 def to\_dict(self):  
 *"""Сериализация избранного города"""* return {  
 'user\_id': self.user\_id,  
 'city\_id': self.city\_id,  
 'created\_at': self.created\_at.isoformat() if self.created\_at else None  
 }  
  
  
def create\_test\_user(email=None, password=None, username=None, role='user'):  
 *"""  
 Создает тестового пользователя с хешированием пароля  
 :param email: Email пользователя (обязательный)  
 :param password: Пароль пользователя (обязательный)  
 :param username: Имя пользователя (обязательный)  
 :param role: Роль пользователя (по умолчанию 'user')  
 :return: Созданный пользователь или None в случае ошибки  
 """* try:  
 # Проверка обязательных полей  
 if not all([email, password, username]):  
 raise ValueError("Email, password and username are required")  
  
 # Проверка допустимых ролей  
 valid\_roles = ['user', 'admin', 'moderator']  
 if role not in valid\_roles:  
 raise ValueError(f"Invalid role. Allowed roles: {', '.join(valid\_roles)}")  
  
 # Проверка существования пользователя  
 if Users.query.filter((Users.email == email) | (Users.username == username)).first():  
 return None  
  
 # Создание пользователя  
 user = Users(  
 email=email,  
 username=username,  
 role=role  
 )  
 user.set\_password(password)  
  
 db.session.add(user)  
 db.session.commit()  
 return user  
  
 except ValueError as ve:  
 print(f"Validation error: {str(ve)}")  
 return None  
 except IntegrityError:  
 db.session.rollback()  
 print("User with this email or username already exists")  
 return None  
 except Exception as e:  
 db.session.rollback()  
 print(f"Error creating test user: {str(e)}")  
 return None  
  
  
def create\_test\_city():  
 *"""Создает тестовый город"""* try:  
 city = Cities(  
 name='Test City',  
 country='Test Country',  
 latitude=0.0,  
 longitude=0.0  
 )  
 db.session.add(city)  
 db.session.commit()  
 return city  
 except Exception as e:  
 db.session.rollback()  
 print(f"Error creating test city: {str(e)}")  
 return None  
  
  
def initialize\_data():  
 *"""Инициализирует тестовые данные"""* try:  
 # Создаем тестовых пользователей  
 users = [  
 create\_test\_user(  
 email="admin@example.com",  
 password="password132",  
 username="admin",  
 role="admin"  
 ),  
 create\_test\_user(  
 email="user1@example.com",  
 password="password1",  
 username="user1",  
 role="user"  
 ),  
 create\_test\_user(  
 email="user2@example.com",  
 password="password2",  
 username="user2",  
 role="user"  
 )  
 ]  
  
 # Создаем тестовые города  
 cities = [  
 create\_test\_city(),  
 Cities(  
 name="Another City",  
 country="Another Country",  
 latitude=10.0,  
 longitude=10.0  
 )  
 ]  
  
 db.session.add\_all(cities)  
 db.session.commit()  
  
 # Добавляем тестовые данные о погоде  
 weather\_data = [  
 WeatherData(  
 city\_id=cities[0].id,  
 temperature=25.0,  
 humidity=50,  
 wind\_speed=5.0,  
 description="Sunny"  
 ),  
 WeatherData(  
 city\_id=cities[1].id,  
 temperature=15.0,  
 humidity=70,  
 wind\_speed=10.0,  
 description="Cloudy"  
 )  
 ]  
  
 db.session.add\_all(weather\_data)  
 db.session.commit()  
  
 print("Test data initialized successfully")  
 return True  
  
 except Exception as e:  
 db.session.rollback()  
 print(f"Error initializing test data: {str(e)}")  
 return False

Код программы routs.py:

from flask import current\_app, jsonify, request  
from flask\_jwt\_extended import get\_jwt\_identity, unset\_jwt\_cookies, create\_access\_token, set\_access\_cookies, \  
 create\_refresh\_token, jwt\_required, get\_jwt, verify\_jwt\_in\_request  
from . import bp # Импортируем Blueprint из текущего модуля  
from backend.src.run import db  
from backend.src.databases.models import Users, Cities, WeatherData, FavoriteCities, create\_test\_user, TokenBlocklist  
import requests  
from datetime import datetime, timedelta, timezone  
from .weather\_service import WeatherService  
from sqlalchemy import func  
from functools import wraps  
from sqlalchemy.exc import IntegrityError  
  
# Конфигурация для внешнего API погоды  
WEATHER\_API\_KEY = 'f4cb5ca908c4c3bfa0bcfa46ec7990b1'  
WEATHER\_API\_URL = 'https://api.openweathermap.org/data/2.5'  
  
def role\_required(required\_role):  
 def decorator(f):  
 @wraps(f)  
 def wrapped(\*args, \*\*kwargs):  
 verify\_jwt\_in\_request()  
 claims = get\_jwt()  
 user\_role = claims.get('role', 'user')  
 if user\_role != required\_role:  
 return jsonify({"error": "Insufficient permissions"}), 403  
 return f(\*args, \*\*kwargs)  
 return wrapped  
 return decorator  
  
admin\_required = role\_required('admin')  
moderator\_required = role\_required('moderator')  
  
# Все роуты должны использовать @bp.route вместо @app.route  
@bp.route('/')  
def home():  
 return "Welcome to the Home Page!"  
  
@bp.route('/users/<int:user\_id>', methods=['DELETE'])  
@admin\_required  
def delete\_user(user\_id):  
 *"""Удаление пользователя по ID"""* try:  
 # Находим пользователя  
 user = Users.query.get(user\_id)  
 if not user:  
 return jsonify({"error": "User not found"}), 404  
  
 # Удаляем связанные записи в favorite\_cities (чтобы избежать ошибок внешнего ключа)  
 FavoriteCities.query.filter\_by(user\_id=user\_id).delete()  
  
 # Удаляем самого пользователя  
 db.session.delete(user)  
 db.session.commit()  
  
 return jsonify({  
 "message": "User deleted successfully",  
 "deleted\_user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email  
 }  
 }), 200  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({"error": f"Failed to delete user: {str(e)}"}), 500  
  
@bp.route('/users/<int:user\_id>', methods=['GET'])  
@jwt\_required()  
def get\_user(user\_id):  
 *"""Получение информации о пользователе (только для себя) с любимыми городами"""* try:  
 # Получаем текущего пользователя из JWT токена  
 current\_user\_id = int(get\_jwt\_identity())  
 current\_user = Users.query.get(current\_user\_id)  
  
 # Проверяем существование текущего пользователя  
 if not current\_user:  
 return jsonify({"error": "Current user not found"}), 404  
  
 # Проверяем, что пользователь запрашивает свои данные  
 if current\_user.id != user\_id:  
 return jsonify({"error": "You can only view your own profile"}), 403  
  
 # Получаем информацию о пользователе  
 user = Users.query.get(user\_id)  
 if not user:  
 return jsonify({"error": "User not found"}), 404  
  
 # Получаем список любимых городов  
 favorites = FavoriteCities.query.filter\_by(user\_id=user\_id).all()  
 favorite\_cities = []  
  
 for fav in favorites:  
 city = Cities.query.get(fav.city\_id)  
 if city:  
 favorite\_cities.append(city.to\_dict())  
  
 return jsonify({  
 "user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email,  
 "created\_at": user.created\_at.isoformat() if user.created\_at else None,  
 "last\_login": user.last\_login.isoformat() if user.last\_login else None  
 },  
 "favorite\_cities": favorite\_cities  
 }), 200  
  
 except ValueError:  
 return jsonify({"error": "Invalid user ID format in token"}), 400  
 except Exception as e:  
 return jsonify({"error": str(e)}), 500  
  
  
@bp.route('/users/<int:user\_id>/favorites/weather', methods=['GET'])  
@jwt\_required()  
def get\_favorite\_cities\_weather(user\_id):  
 *"""Получение текущей погоды для любимых городов пользователя (только для себя)"""* try:  
 # Получаем текущего пользователя из JWT токена  
 current\_user\_id = int(get\_jwt\_identity())  
  
 # Проверяем, что пользователь запрашивает свои данные  
 if current\_user\_id != user\_id:  
 return jsonify({"error": "You can only view weather for your own favorite cities"}), 403  
  
 # Получаем пользователя из БД  
 user = Users.query.get(user\_id)  
 if not user:  
 return jsonify({"error": "User not found"}), 404  
  
 # Получаем список любимых городов пользователя с join для оптимизации запроса  
 favorites = db.session.query(  
 FavoriteCities,  
 Cities,  
 WeatherData  
 ).join(  
 Cities, FavoriteCities.city\_id == Cities.id  
 ).join(  
 WeatherData, Cities.id == WeatherData.city\_id  
 ).filter(  
 FavoriteCities.user\_id == user\_id  
 ).order\_by(  
 WeatherData.timestamp.desc()  
 ).all()  
  
 # Группируем по городам и берем последнюю запись о погоде для каждого  
 weather\_data = []  
 processed\_cities = set()  
  
 for fav, city, weather in favorites:  
 if city.id not in processed\_cities:  
 weather\_data.append({  
 "city": city.to\_dict(),  
 "weather": weather.to\_dict()  
 })  
 processed\_cities.add(city.id)  
  
 return jsonify({  
 "message": "Weather for favorite cities retrieved successfully",  
 "count": len(weather\_data),  
 "favorite\_cities\_weather": weather\_data  
 }), 200  
  
 except ValueError:  
 return jsonify({"error": "Invalid user ID format in token"}), 400  
 except Exception as e:  
 return jsonify({"error": f"Failed to get weather data: {str(e)}"}), 500  
  
@bp.route('/users/search', methods=['GET'])  
@admin\_required  
def search\_users():  
 *"""Поиск пользователей по имени или email"""* username = request.args.get('username')  
 email = request.args.get('email')  
  
 query = Users.query  
  
 if username:  
 query = query.filter(Users.username.ilike(f'%{username}%'))  
 if email:  
 query = query.filter(Users.email.ilike(f'%{email}%'))  
  
 users = query.all()  
  
 return jsonify([{  
 "id": u.id,  
 "username": u.username,  
 "email": u.email  
 } for u in users]), 200  
  
@bp.route('/cities', methods=['POST'])  
@admin\_required  
def add\_city():  
 *"""Добавление нового города в базу данных по названию (данные берутся из OpenWeatherMap)"""* try:  
 # Получаем данные из запроса  
 data = request.get\_json()  
  
 # Проверяем обязательное поле - название города  
 if not data or 'name' not in data or not data['name'].strip():  
 return jsonify({  
 "error": "City name is required and cannot be empty",  
 "required\_fields": ["name"]  
 }), 400  
  
 city\_name = data['name'].strip()  
  
 # Проверяем, существует ли уже город с таким названием (регистронезависимо)  
 existing\_city = Cities.query.filter(  
 func.lower(Cities.name) == func.lower(city\_name)  
 ).first()  
  
 if existing\_city:  
 return jsonify({  
 "error": "City already exists",  
 "existing\_city": {  
 "id": existing\_city.id,  
 "name": existing\_city.name,  
 "country": existing\_city.country  
 }  
 }), 409  
  
 # Получаем данные о городе напрямую из OpenWeatherMap API  
 geo\_url = "http://api.openweathermap.org/geo/1.0/direct"  
 params = {  
 'q': city\_name,  
 'limit': 1,  
 'appid': WEATHER\_API\_KEY # Ваш API ключ  
 }  
  
 response = requests.get(geo\_url, params=params)  
  
 # Обрабатываем возможные ошибки API  
 if response.status\_code != 200:  
 return jsonify({  
 "error": "OpenWeatherMap API unavailable",  
 "details": f"API returned status {response.status\_code}"  
 }), 502  
  
 data = response.json()  
 if not data:  
 return jsonify({  
 "error": f"City '{city\_name}' not found in OpenWeatherMap"  
 }), 404  
  
 city\_info = data[0]  
  
 # Создаем новый город  
 new\_city = Cities(  
 name=city\_info.get('name'),  
 country=city\_info.get('country'),  
 latitude=city\_info.get('lat'),  
 longitude=city\_info.get('lon')  
 )  
  
 db.session.add(new\_city)  
 db.session.commit()  
  
 return jsonify({  
 "message": "City added successfully",  
 "city": {  
 "id": new\_city.id,  
 "name": new\_city.name,  
 "country": new\_city.country,  
 "coordinates": {  
 "latitude": new\_city.latitude,  
 "longitude": new\_city.longitude  
 }  
 }  
 }), 201  
  
 except ValueError as e:  
 db.session.rollback()  
 return jsonify({  
 "error": "Invalid coordinate values",  
 "details": str(e)  
 }), 400  
  
 except Exception as e:  
 db.session.rollback()  
 current\_app.logger.error(f"Failed to add city: {str(e)}")  
 return jsonify({  
 "error": "Failed to add city",  
 "details": str(e)  
 }), 500  
  
@bp.route('/cities/<int:city\_id>', methods=['DELETE'])  
@admin\_required  
def delete\_city(city\_id):  
 *"""Удаление города из базы данных по ID"""* try:  
 # Находим город  
 city = Cities.query.get(city\_id)  
 if not city:  
 return jsonify({"error": "City not found"}), 404  
  
 # Проверяем, есть ли связанные записи в таблицах  
 weather\_records = WeatherData.query.filter\_by(city\_id=city\_id).count()  
 favorite\_records = FavoriteCities.query.filter\_by(city\_id=city\_id).count()  
  
 if weather\_records > 0 or favorite\_records > 0:  
 return jsonify({  
 "error": "Cannot delete city - it has related records",  
 "details": {  
 "weather\_records": weather\_records,  
 "favorite\_records": favorite\_records  
 }  
 }), 409  
  
 # Удаляем город  
 db.session.delete(city)  
 db.session.commit()  
  
 return jsonify({  
 "message": "City deleted successfully",  
 "deleted\_city": {  
 "id": city.id,  
 "name": city.name,  
 "country": city.country  
 }  
 }), 200  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({  
 "error": "Failed to delete city",  
 "details": str(e)  
 }), 500  
  
  
@bp.route('/users/<int:user\_id>/favorites', methods=['POST'])  
@jwt\_required()  
def add\_favorite\_city(user\_id):  
 *"""Добавление любимого города (только для своего аккаунта)"""* try:  
 # Приводим ID к одному типу (str или int)  
 current\_user\_id = int(get\_jwt\_identity()) # Преобразуем в int  
  
 if current\_user\_id != user\_id:  
 return jsonify({"error": "You can only add favorites to your own account"}), 403  
  
 # Остальной код без изменений...  
 data = request.get\_json()  
 if not data or 'city\_id' not in data:  
 return jsonify({"error": "City ID is required"}), 400  
  
 city\_id = data['city\_id']  
 city = Cities.query.get(city\_id)  
 if not city:  
 return jsonify({"error": "City not found"}), 404  
  
 if FavoriteCities.query.filter\_by(user\_id=user\_id, city\_id=city\_id).first():  
 return jsonify({  
 "error": "City already in favorites",  
 "city": city.to\_dict()  
 }), 409  
  
 new\_favorite = FavoriteCities(user\_id=user\_id, city\_id=city\_id)  
 db.session.add(new\_favorite)  
 db.session.commit()  
  
 return jsonify({  
 "message": "City added to favorites",  
 "city": city.to\_dict()  
 }), 201  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({"error": str(e)}), 500  
  
@bp.route('/users/<int:user\_id>/favorites/<int:city\_id>', methods=['DELETE'])  
@jwt\_required()  
def remove\_favorite\_city(user\_id, city\_id):  
 *"""Удаление города из избранного (только для своего аккаунта)"""* try:  
 # Проверка прав доступа с приведением типов  
 current\_user\_id = int(get\_jwt\_identity()) # Преобразуем строку в число  
 if current\_user\_id != user\_id:  
 return jsonify({"error": "You can only remove favorites from your own account"}), 403  
  
 # Ищем связь пользователя с городом  
 favorite = FavoriteCities.query.filter\_by(  
 user\_id=user\_id,  
 city\_id=city\_id  
 ).first()  
  
 if not favorite:  
 return jsonify({"error": "Favorite city not found"}), 404  
  
 # Удаляем  
 db.session.delete(favorite)  
 db.session.commit()  
  
 return jsonify({  
 "message": "City removed from favorites",  
 "removed\_city": {  
 "city\_id": city\_id,  
 "user\_id": user\_id  
 }  
 }), 200  
  
 except ValueError:  
 return jsonify({"error": "Invalid user ID format in token"}), 400  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({"error": str(e)}), 500  
  
@bp.route('/weather/<int:city\_id>', methods=['GET'])  
def get\_city\_weather(city\_id):  
 *"""Получение всех записей о погоде для указанного города по ID"""* try:  
 # Проверяем существование города  
 city = Cities.query.get(city\_id)  
 if not city:  
 return jsonify({"error": f"City with ID {city\_id} not found"}), 404  
  
 # Получаем все записи о погоде для города, отсортированные по времени (новые сначала)  
 weather\_records = WeatherData.query.filter\_by(city\_id=city\_id) \  
 .order\_by(WeatherData.timestamp.desc()) \  
 .all()  
  
 if not weather\_records:  
 # Если данных нет в БД, пытаемся получить свежие  
 WeatherService.save\_weather\_data(city.name)  
 weather\_records = WeatherData.query.filter\_by(city\_id=city\_id) \  
 .order\_by(WeatherData.timestamp.desc()) \  
 .all()  
 if not weather\_records:  
 return jsonify({"error": "No weather data found for this city"}), 404  
  
 # Формируем ответ  
 response = {  
 "city": {  
 "id": city.id,  
 "name": city.name,  
 "country": city.country,  
 "coordinates": {  
 "latitude": city.latitude,  
 "longitude": city.longitude  
 }  
 },  
 "weather\_data": [{  
 "temperature": record.temperature,  
 "humidity": record.humidity,  
 "wind\_speed": record.wind\_speed,  
 "description": record.description,  
 "timestamp": record.timestamp.isoformat() if record.timestamp else None  
 } for record in weather\_records]  
 }  
  
 return jsonify(response), 200  
  
 except Exception as e:  
 return jsonify({"error": f"Failed to get weather data: {str(e)}"}), 500  
  
@bp.route('/weather/update\_hourly/<int:city\_id>', methods=['POST'])  
@admin\_required  
def update\_hourly\_weather(city\_id):  
 *"""  
 Добавляет почасовые данные о погоде для указанного города по ID:  
 - За предыдущие 12 часов  
 - Текущий час  
 - Следующие 12 часов  
 Всего 25 записей  
 """* try:  
 # Проверяем существование города  
 city = Cities.query.get(city\_id)  
 if not city:  
 return jsonify({"error": f"City with ID {city\_id} not found"}), 404  
  
 # 1. Получаем прогноз на 24 часа (12 прошедших + 12 будущих + текущий)  
 forecast = WeatherService.get\_24h\_forecast(city.name)  
 if not forecast:  
 return jsonify({"error": f"Failed to get 24-hour forecast for city ID {city\_id}"}), 500  
  
 # 2. Добавляем все записи о погоде  
 added\_records = []  
 for weather\_data in forecast:  
 # Проверяем, нет ли уже такой записи  
 existing = WeatherData.query.filter\_by(  
 city\_id=city\_id,  
 timestamp=weather\_data['timestamp']  
 ).first()  
  
 if not existing:  
 weather\_record = WeatherData(  
 city\_id=city\_id,  
 temperature=weather\_data['temperature'],  
 humidity=weather\_data['humidity'],  
 wind\_speed=weather\_data['wind\_speed'],  
 description=weather\_data['description'],  
 timestamp=weather\_data['timestamp']  
 )  
 db.session.add(weather\_record)  
 added\_records.append({  
 'timestamp': weather\_data['timestamp'].isoformat(),  
 'temperature': weather\_data['temperature']  
 })  
  
 db.session.commit()  
  
 return jsonify({  
 "message": f"Hourly weather data updated for city ID {city\_id}",  
 "city\_id": city\_id,  
 "city\_name": city.name,  
 "added\_records": added\_records,  
 "total\_added": len(added\_records),  
 "time\_range": {  
 "start": forecast[0]['timestamp'].isoformat(),  
 "end": forecast[-1]['timestamp'].isoformat()  
 }  
 }), 200  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({  
 "error": "Failed to update hourly weather data",  
 "details": str(e)  
 }), 500  
  
@bp.route('/weather/cleanup', methods=['DELETE'])  
@admin\_required  
def cleanup\_old\_weather\_data():  
 *"""  
 Удаляет устаревшие записи о погоде:  
 - Старше 1 день для всех городов  
 - Оставляет минимум 24 записи для каждого города  
 Использует datetime.now(timezone.utc) вместо устаревшего utcnow()  
 """* try:  
 # Определяем временную границу (7 дней назад)  
 seven\_days\_ago = datetime.now(timezone.utc) - timedelta(days=1)  
  
 # Сначала находим города, у которых больше 24 записей  
 cities\_to\_clean = db.session.query(  
 WeatherData.city\_id  
 ).group\_by(  
 WeatherData.city\_id  
 ).having(  
 db.func.count(WeatherData.id) > 24  
 ).all()  
  
 total\_deleted = 0  
  
 # Для каждого города удаляем старые записи, оставляя минимум 24 самых свежих  
 for city in cities\_to\_clean:  
 city\_id = city[0]  
  
 # Находим ID 24 самых свежих записей  
 latest\_ids = [  
 w[0] for w in db.session.query(  
 WeatherData.id  
 ).filter(  
 WeatherData.city\_id == city\_id  
 ).order\_by(  
 WeatherData.timestamp.desc()  
 ).limit(24).all()  
 ]  
  
 # Удаляем записи старше 7 дней, кроме 24 последних  
 deleted = WeatherData.query.filter(  
 WeatherData.city\_id == city\_id,  
 WeatherData.timestamp < seven\_days\_ago,  
 ~WeatherData.id.in\_(latest\_ids)  
 ).delete(synchronize\_session=False)  
  
 total\_deleted += deleted  
  
 db.session.commit()  
  
 return jsonify({  
 "message": "Old weather data cleanup completed",  
 "details": {  
 "cities\_processed": len(cities\_to\_clean),  
 "records\_deleted": total\_deleted,  
 "cutoff\_date": seven\_days\_ago.isoformat()  
 }  
 }), 200  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({  
 "error": "Failed to cleanup old weather data",  
 "details": str(e)  
 }), 500  
  
# Регистрация пользователя  
@bp.route('/register', methods=['POST'])  
def register():  
 import re # Импортируем модуль для работы с регулярными выражениями  
  
 data = request.get\_json()  
 required\_fields = ['email', 'password', 'username']  
  
 if not data or not all(key in data for key in required\_fields):  
 return jsonify({"error": "Missing required fields"}), 400  
  
 # Валидация email  
 email\_regex = r'^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'  
 if not re.match(email\_regex, data['email']):  
 return jsonify({"error": "Invalid email format"}), 400  
  
 if Users.query.filter\_by(email=data['email']).first():  
 return jsonify({"error": "Email already registered"}), 400  
  
 if Users.query.filter\_by(username=data['username']).first():  
 return jsonify({"error": "Username already taken"}), 400  
  
 try:  
 # Создаем пользователя с ролью по умолчанию 'user'  
 user = Users(  
 email=data['email'],  
 username=data['username'],  
 role=data.get('role', 'user') # Можно задать роль при регистрации, если нужно  
 )  
 user.set\_password(data['password'])  
 db.session.add(user)  
 db.session.commit()  
  
 # Создаем токены с дополнительными claims (включая роль)  
 additional\_claims = {"role": user.role}  
 access\_token = create\_access\_token(  
 identity=str(user.id),  
 additional\_claims=additional\_claims  
 )  
 refresh\_token = create\_refresh\_token(  
 identity=str(user.id),  
 additional\_claims=additional\_claims  
 )  
  
 return jsonify({  
 "message": "User registered successfully",  
 "access\_token": access\_token,  
 "refresh\_token": refresh\_token,  
 "user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email,  
 "role": user.role # Добавляем роль в ответ  
 }  
 }), 201  
  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({"error": str(e)}), 500  
  
  
@bp.route('/login', methods=['POST'])  
def login():  
 data = request.get\_json()  
 if not data or 'email' not in data or 'password' not in data:  
 return jsonify({"error": "Email and password required"}), 400  
  
 user = Users.query.filter\_by(email=data['email']).first()  
 if not user or not user.check\_password(data['password']):  
 return jsonify({"error": "Invalid credentials"}), 401  
  
 # Создаем токены с дополнительными claims (включая роль)  
 additional\_claims = {"role": user.role}  
 access\_token = create\_access\_token(  
 identity=str(user.id),  
 additional\_claims=additional\_claims  
 )  
 refresh\_token = create\_refresh\_token(  
 identity=str(user.id),  
 additional\_claims=additional\_claims  
 )  
  
 return jsonify({  
 "message": "Login successful",  
 "access\_token": access\_token,  
 "refresh\_token": refresh\_token,  
 "user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email,  
 "role": user.role # Добавляем роль в ответ  
 }  
 }), 200  
  
# Обновление токена  
@bp.route('/refresh', methods=['POST'])  
@jwt\_required(refresh=True)  
def refresh():  
 # Получаем данные из текущего refresh-токена  
 current\_user = get\_jwt\_identity()  
 claims = get\_jwt()  
  
 # Проверяем, есть ли роль в claims (на случай устаревших токенов)  
 user\_role = claims.get('role', 'user')  
  
 # Создаем новый access-токен с сохранением роли  
 new\_token = create\_access\_token(  
 identity=str(current\_user),  
 additional\_claims={'role': user\_role}  
 )  
  
 return jsonify({  
 "access\_token": new\_token,  
 "role": user\_role # Для удобства фронтенда  
 }), 200  
  
# Выход из системы  
@bp.route('/logout', methods=['POST'])  
@jwt\_required()  
def logout():  
 jti = get\_jwt()["jti"]  
 now = datetime.now(timezone.utc)  
 db.session.add(TokenBlocklist(jti=jti, created\_at=now))  
 db.session.commit()  
 return jsonify({"message": "Successfully logged out"}), 200  
  
  
@bp.route('/verify', methods=['GET'])  
@jwt\_required()  
def verify\_token():  
 try:  
 # Получаем ID из токена (теперь будет работать корректно)  
 user\_id = get\_jwt\_identity()  
 user = Users.query.get(user\_id)  
  
 if not user:  
 return jsonify({"error": "User not found"}), 404  
  
 return jsonify({  
 "message": "Token is valid",  
 "user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email  
 }  
 }), 200  
  
 except Exception as e:  
 return jsonify({"error": "Token verification failed", "details": str(e)}), 401  
  
#Изменение пароля  
@bp.route('/update-account', methods=['POST'])  
@jwt\_required()  
def update\_account():  
 current\_user\_id = get\_jwt\_identity()  
 data = request.get\_json()  
  
 # Базовые проверки  
 if not data:  
 return jsonify({"error": "No data provided"}), 400  
  
 user = Users.query.get(current\_user\_id)  
 if not user:  
 return jsonify({"error": "User not found"}), 404  
  
 # Изменение пароля (требуется старый пароль)  
 if 'new\_password' in data:  
 if 'old\_password' not in data:  
 return jsonify({"error": "Old password is required to set new password"}), 400  
  
 if not user.check\_password(data['old\_password']):  
 return jsonify({"error": "Invalid old password"}), 401  
  
 if len(data['new\_password']) < 8:  
 return jsonify({"error": "New password must be at least 8 characters"}), 400  
  
 user.set\_password(data['new\_password'])  
  
 # Изменение email  
 if 'email' in data:  
 new\_email = data['email'].strip()  
 if not new\_email:  
 return jsonify({"error": "Email cannot be empty"}), 400  
  
 if new\_email != user.email and Users.query.filter\_by(email=new\_email).first():  
 return jsonify({"error": "Email already in use"}), 400  
  
 user.email = new\_email  
  
 # Изменение username  
 if 'username' in data:  
 new\_username = data['username'].strip()  
 if not new\_username:  
 return jsonify({"error": "Username cannot be empty"}), 400  
  
 if new\_username != user.username and Users.query.filter\_by(username=new\_username).first():  
 return jsonify({"error": "Username already taken"}), 400  
  
 user.username = new\_username  
  
 # Если нет изменений  
 if not any(field in data for field in ['new\_password', 'email', 'username']):  
 return jsonify({"error": "No fields to update provided"}), 400  
  
 try:  
 db.session.commit()  
 return jsonify({  
 "message": "Account updated successfully",  
 "user": {  
 "id": user.id,  
 "username": user.username,  
 "email": user.email  
 }  
 }), 200  
 except Exception as e:  
 db.session.rollback()  
 return jsonify({"error": str(e)}), 500  
  
@bp.route('/server\_time', methods=['GET'])  
def get\_server\_time():  
 return jsonify({  
 "server\_time\_utc": datetime.utcnow().isoformat(),  
 "server\_time\_local": datetime.now().isoformat()  
 }), 200

Код файла weather\_service.py:

import requests  
from datetime import datetime  
from backend.src.run import db  
from backend.src.databases.models import Cities, WeatherData  
  
class WeatherService:  
 API\_KEY = 'f4cb5ca908c4c3bfa0bcfa46ec7990b1' # Ваш API ключ  
 BASE\_URL = 'https://api.openweathermap.org/data/2.5'  
  
 @classmethod  
 def get\_current\_weather(cls, city\_name):  
 *"""Получает текущую погоду для указанного города"""* url = f"{cls.BASE\_URL}/weather?q={city\_name}&appid={cls.API\_KEY}&units=metric"  
 response = requests.get(url)  
  
 if response.status\_code != 200:  
 return None  
  
 return cls.\_parse\_weather\_data(response.json())  
  
 @classmethod  
 def get\_forecast(cls, city\_name, days=5):  
 *"""Получает прогноз погоды на несколько дней"""* url = f"{cls.BASE\_URL}/forecast?q={city\_name}&appid={cls.API\_KEY}&units=metric&cnt={days \* 8}"  
 response = requests.get(url)  
  
 if response.status\_code != 200:  
 return None  
  
 return [cls.\_parse\_weather\_data(item) for item in response.json().get('list', [])]  
  
 @staticmethod  
 def \_parse\_weather\_data(data):  
 *"""Парсит данные о погоде в унифицированный формат"""* return {  
 "city\_id": None, # Будет заполнено при сохранении в БД  
 "temperature": data.get('main', {}).get('temp'),  
 "humidity": data.get('main', {}).get('humidity'),  
 "wind\_speed": data.get('wind', {}).get('speed'),  
 "description": data['weather'][0]['description'] if data.get('weather') else None,  
 "timestamp": datetime.fromtimestamp(data.get('dt')) if data.get('dt') else None,  
 "raw\_data": data # Сохраняем исходные данные на всякий случай  
 }  
  
 @classmethod  
 def save\_weather\_data(cls, city\_name):  
 *"""Получает и сохраняет данные о погоде в БД"""* # Проверяем, есть ли город в БД  
 city = Cities.query.filter\_by(name=city\_name).first()  
 if not city:  
 # Если города нет, сначала получаем его координаты  
 weather\_data = cls.get\_current\_weather(city\_name)  
 if not weather\_data:  
 return None  
  
 # Создаем новую запись города  
 city = Cities(  
 name=city\_name,  
 country=weather\_data['raw\_data'].get('sys', {}).get('country'),  
 latitude=weather\_data['raw\_data'].get('coord', {}).get('lat'),  
 longitude=weather\_data['raw\_data'].get('coord', {}).get('lon')  
 )  
 db.session.add(city)  
 db.session.flush()  
  
 # Получаем текущую погоду  
 weather\_data = cls.get\_current\_weather(city\_name)  
 if not weather\_data:  
 return None  
  
 # Создаем запись о погоде  
 weather = WeatherData(  
 city\_id=city.id,  
 temperature=weather\_data['temperature'],  
 humidity=weather\_data['humidity'],  
 wind\_speed=weather\_data['wind\_speed'],  
 description=weather\_data['description'],  
 timestamp=weather\_data['timestamp']  
 )  
 db.session.add(weather)  
 db.session.commit()  
  
 return weather  
  
 @classmethod  
 def get\_24h\_forecast(cls, city\_name):  
 *"""Получает почасовой прогноз на 24 часа (включая текущий)"""* url = f"{cls.BASE\_URL}/forecast?q={city\_name}&appid={cls.API\_KEY}&units=metric&cnt=25"  
 response = requests.get(url)  
  
 if response.status\_code != 200:  
 return None  
  
 forecast\_data = response.json()['list']  
  
 # Обрабатываем данные для 25 периодов (24 часа + текущий)  
 forecast = []  
 for item in forecast\_data[:25]: # Берем только 25 записей  
 parsed = cls.\_parse\_weather\_data(item)  
 forecast.append(parsed)  
  
 return forecast  
  
 def get\_city\_data(self, city\_name):  
 *"""Получение данных о городе из OpenWeatherMap"""* url = f"{self.base\_url}/geo/1.0/direct"  
 params = {  
 'q': city\_name,  
 'limit': 1,  
 'appid': self.api\_key  
 }  
 response = requests.get(url, params=params)  
 if response.status\_code == 200 and response.json():  
 data = response.json()[0]  
 return {  
 'name': data.get('name'),  
 'country': data.get('country'),  
 'lat': data.get('lat'),  
 'lon': data.get('lon')  
 }  
 return None

Код файла AuthForm.js:

import React, { useState } from 'react';  
import '../styles/AuthForm.css';  
  
function AuthForm({ isLogin, onSubmit, disabled }) {  
 const [formData, setFormData] = useState({  
 username: '',  
 email: '',  
 emailOrUsername: '',  
 password: ''  
 });  
 const [errors, setErrors] = useState({});  
  
 const handleChange = (e) => {  
 const { name, value } = e.target;  
 setFormData(prev => ({ ...prev, [name]: value }));  
 if (errors[name]) {  
 setErrors(prev => ({ ...prev, [name]: '' }));  
 }  
 };  
  
 const validateForm = () => {  
 const newErrors = {};  
  
 if (isLogin) {  
 if (!formData.emailOrUsername.trim()) {  
 newErrors.emailOrUsername = 'Введите email или имя пользователя';  
 }  
 } else {  
 if (!formData.username.trim()) {  
 newErrors.username = 'Введите имя пользователя';  
 }  
 if (!formData.email.trim() || !/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(formData.email)) {  
 newErrors.email = 'Введите корректный email';  
 }  
 }  
  
 if (!formData.password || formData.password.length < 6) {  
 newErrors.password = 'Пароль должен содержать минимум 6 символов';  
 }  
  
 setErrors(newErrors);  
 return Object.keys(newErrors).length === 0;  
 };  
  
 const handleSubmit = (e) => {  
 e.preventDefault();  
 if (!validateForm()) return;  
  
 const data = isLogin  
 ? {  
 emailOrUsername: formData.emailOrUsername.trim(),  
 password: formData.password  
 }  
 : {  
 username: formData.username.trim(),  
 email: formData.email.trim(),  
 password: formData.password  
 };  
  
 onSubmit(data);  
 };  
  
 return (  
 <form onSubmit={handleSubmit} className="auth-form" autoComplete="off">  
 {!isLogin ? (  
 <>  
 <div className="form-group">  
 <input  
 type="text"  
 name="username"  
 placeholder="Имя пользователя"  
 value={formData.username}  
 onChange={handleChange}  
 disabled={disabled}  
 className={errors.username ? 'error' : ''}  
 />  
 {errors.username && <span className="error-text">{errors.username}</span>}  
 </div>  
 <div className="form-group">  
 <input  
 type="email"  
 name="email"  
 placeholder="Email"  
 value={formData.email}  
 onChange={handleChange}  
 disabled={disabled}  
 className={errors.email ? 'error' : ''}  
 />  
 {errors.email && <span className="error-text">{errors.email}</span>}  
 </div>  
 </>  
 ) : (  
 <div className="form-group">  
 <input  
 type="text"  
 name="emailOrUsername"  
 placeholder="Email или имя пользователя"  
 value={formData.emailOrUsername}  
 onChange={handleChange}  
 disabled={disabled}  
 className={errors.emailOrUsername ? 'error' : ''}  
 />  
 {errors.emailOrUsername && <span className="error-text">{errors.emailOrUsername}</span>}  
 </div>  
 )}  
 <div className="form-group">  
 <input  
 type="password"  
 name="password"  
 placeholder="Пароль"  
 value={formData.password}  
 onChange={handleChange}  
 disabled={disabled}  
 className={errors.password ? 'error' : ''}  
 />  
 {errors.password && <span className="error-text">{errors.password}</span>}  
 </div>  
 <button type="submit" disabled={disabled}>  
 {isLogin ? 'Войти' : 'Зарегистрироваться'}  
 </button>  
 </form>  
 );  
}  
  
export default AuthForm;

Код файла Navbar.js:

import React from 'react';  
import { useNavigate } from 'react-router-dom';  
import { authFetch, clearAuth } from '../api/http';  
import '../styles/main.css';  
  
function Navbar({ user, setUser }) {  
 const navigate = useNavigate();  
  
 const handleLogout = async () => {  
 try {  
 await authFetch('/logout', { method: 'POST' });  
 } catch (error) {  
 console.error('Logout error:', error);  
 } finally {  
 clearAuth();  
 setUser(null);  
 navigate('/auth');  
 }  
 };  
  
 return (  
 <nav>  
 <button onClick={() => navigate('/')}>Главная</button>  
 {user ? (  
 <>  
 <button onClick={() => navigate('/profile')}>Профиль</button>  
 <button onClick={handleLogout}>Выйти</button>  
 </>  
 ) : (  
 <button onClick={() => navigate('/auth')}>Войти</button>  
 )}  
 </nav>  
 );  
}  
  
export default Navbar;

Код файла WeatherCard.js:

import React from 'react';  
import { FaStar, FaRegStar } from 'react-icons/fa';  
import '../styles/WeatherCard.css';  
  
function WeatherCard({ data, onToggleFavorite, isFavorite }) {  
 const formatDate = (timestamp) => {  
 if (!timestamp) return '';  
 const date = new Date(timestamp);  
 return date.toLocaleString();  
 };  
  
 return (  
 <div className="weather-card">  
 <div className="card-header">  
 <h3>{data.city}</h3>  
 {onToggleFavorite && (  
 <button  
 className="favorite-btn"  
 onClick={(e) => {  
 e.stopPropagation();  
 onToggleFavorite(data.id);  
 }}  
 aria-label={isFavorite ? "Удалить из избранного" : "Добавить в избранное"}  
 >  
 {isFavorite ? <FaStar color="gold" size={20} /> : <FaRegStar size={20} />}  
 </button>  
 )}  
 </div>  
 <div className="weather-main">  
 <span className="temp">{data.temp}°C</span>  
 <span className="description">{data.description}</span>  
 </div>  
 <div className="weather-details">  
 <p>Влажность: {data.humidity}%</p>  
 <p>Ветер: {data.wind\_speed} м/с</p>  
 {data.timestamp && <p className="timestamp">{formatDate(data.timestamp)}</p>}  
 </div>  
 </div>  
 );  
}  
  
export default WeatherCard;