МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики
Кафедра Математического обеспечения и суперкомпьютерных технологий
Направление подготовки “Инженерия программного обеспечения”

ОТЧЁТ
по учебной практике на тему
“Разработка программно-аппаратного комплекса для мониторинга показателей сердца человека”

Выполнил:
студент группы 382008-1
Булгаков Д.Э.
Подпись

Проверил:
к.т.н., доц.
Борисов Н.А.
Подпись

Нижний Новгород
2024

Содержание

1. Введение

Болезни сердца и сосудов, которые приводят к большому числу смертей по всему миру, привлекают большое внимание медицинского сообщества. Ишемическая болезнь сердца, высокое артериальное давление и другие проблемы с сердечно-сосудистой системой требуют серьезного подхода к их выявлению и профилактике.

Сердечно-сосудистая система, сложная и многофункциональная, подвержена различным воздействиям, которые могут привести к серьезным нарушениям. Регулярные обследования играют ключевую роль в этом контексте. Они не только помогают выявлять проблемы на ранних стадиях, но и открывают пути для раннего вмешательства и эффективной профилактики. Такой подход становится неотъемлемой частью стратегии поддержания здоровья сердечно-сосудистой системы в современном мире.

2. Постановка задачи.

Необходимо создать сервер, способный принимать, хранить и анализировать данные ЭКГ. Сервер будет являться связующим звеном между модулем снятия ЭКГ на основе ESP32 и web-сайтом, на котором пользователи смогут просматривать свои показания и возможные проблемы с сердечно-сосудистой системой.

1. **Выбрать стек технологий, на основе которых будет написан сервер.**
    - Определить язык программирования для сервера.
    - Определить фреймворк для написания сервера.
    - Выбрать подходящую базу данных.
2. **Определить сценарии использования сервера.**
   Сценарии использования разделены на следующие платформы:
    - Web-сайт на основе Vue.js
    - Electron-приложение.

   Общие сценарии можно обобщить и выделить в отдельную компоненту.
3. **Разработать и реализовать архитектуру сервера.**

3. Проведенная работа.

3.1. Стек технологий.

3.1.1. Язык программирования и фреймворк.

Перед началом проектирования архитектуры сервера, требуется выбрать язык программирования и фреймворк, который обеспечивал бы удобные инструменты для обработки HTTP-запросов, маршрутизации и взаимодействия с базой данных. На рассмотрении были следующие варианты:

- Node.js с фреймворком Express.js
- Golang с фреймворками Gin или Martini
- Python с фреймворками Flask или Django

Необходимо сравнить основные характеристики каждого решения, такие как производительность, удобство и простота разработки, доступность библиотек и инструментов, а также сообщество и поддержка. Прежде всего, было решено остановиться на языке программирования Python по следующим причинам:

- **Простота и читаемость кода:**
  Python известен своим чистым и выразительным синтаксисом, который делает код легким для чтения, понимания и поддержки.
- **Широкие возможности для веб-разработки:**
  Python имеет обширную экосистему библиотек и фреймворков для веб-разработки. Flask, Django, FastAPI - все они предоставляют мощные инструменты для создания веб-приложений любого уровня сложности.
- **Кросс-платформенность и портативность:**
  Python работает на множестве операционных систем, что обеспечивает гибкость и портативность разработки. Это означает, что код, написанный на Python, может быть легко перенесен с одной платформы на другую без необходимости внесения значительных изменений.

- **Богатая стандартная библиотека:**
  Python поставляется с обширной стандартной библиотекой, которая включает в себя множество модулей для работы с сетью, обработки данных, взаимодействия с базами данных и многое другое.

Осталось определиться с фреймворком Python. Было решено использовать Flask, т.к. сервер является учебным, небольшим проектом, он не требует таких глобальных и серьезных инструментов, которые предоставляет Django. Также Flask обладает более интуитивно понятный интерфейсом, что позволит меньше времени потратить на изучение инструмента.

3.1.2. База данных.

Для структурирования и хранения данных необходимо выбрать базу данных, которая лучше всего подойдет к выбранному фреймворку. На рассмотрении были следующие варианты:

- PostgreSQL
- SQLite
- MySQL
- MongoDB
- Redis

По итогу, было решено использовать PostgreSQL из-за его надежности, расширяемости, а также доступности. Кроме того, PostgreSQL активно развивается сообществом и имеет обширную документацию, что делает его привлекательным выбором для учебных проектов.

3.1.3. Выбор инструментов для развертывания приложения.

Для обеспечения более удобного и гибкого процесса разработки и развертывания, было принято решение использовать Docker.

Docker обеспечивает создание и управление контейнеризированными средами разработки и развертывания, что значительно упрощает процесс настройки окружения и обеспечивает

переносимость приложения между различными средами. Кроме того, Docker обеспечивает изолированное выполнение приложений, что позволяет избежать конфликтов между зависимостями и обеспечивает надежность работы приложения.

3.2. Сценарии использования.

Серверная часть приложения должна обрабатывать запросы, поступающие от двух типов клиентских приложений:

- Десктопное приложение на основе фреймворка Electron.
- Web-приложение на основе Vue.js

Некоторые сценарии использования применимы к обоим типам приложений, поэтому они выделены в отдельный модуль, который содержит общие сценарии использования.

3.2.1. Общие сценарии использования.

- **Идентификация, аутентификация и авторизация пользователя:**
  На сервере будет хранится и обрабатываться информация всех пользователей, поэтому пользователи должны иметь возможность идентифицировать себя на сервере, проходить аутентификацию и, при необходимости, проходить авторизацию для доступа к определенным ресурсам или функциональности.
- **Маршруты для работы с сессионными ключами:**
  Для работы с цифровыми личностями используются сессионные ключи. Поэтому необходимо поддерживать маршруты для создания новых сессионных ключей, обновление их срока действия и удаление ключей после завершения сеанса работы пользователя.

3.2.2. Сценарии использования для Electron приложения.

Electron-приложение служит посредником для передачи данных ЭКГ с устройства ESP32 на сервер. Кроме общих сценариев использования в API приложения должны быть реализованы следующие пункты:

- **Разработка маршрутов для приема данных ЭКГ от приложения Electron:**
  Предполагает создание маршрутов на сервере, которые будут принимать данные ЭКГ,

отправленные из приложения Electron, и обрабатывать их для последующего сохранения или анализа.

- **Создание механизмов хранения полученных данных ЭКГ на сервере:**
  Требует разработки функционала для сохранения полученных данных ЭКГ на сервере. Это может включать в себя разработку механизмов для хранения и организации данных.

- **Разработка механизмов передачи данных ЭКГ на анализ искусственным интеллектом:**
  Включает в себя создание функционала для передачи данных ЭКГ на анализ искусственным интеллектом. Это может включать в себя разработку API или механизмов интеграции с собственными моделями искусственного интеллекта для обработки данных и генерации результатов анализа.

3.2.3. Сценарии использования для Web-приложения.

Web-приложение является основным источником для изучения и рассмотрения пользователем своих данных ЭКГ. Кроме общих сценариев использования в API приложения должны быть реализованы следующие пункты:

- **Обновление данных пользователя:**
  Пользователь изменяет свои данные, такие как дата рождения, и отправляет их на сервер для обновления.

- **Отображение графиков ЭКГ:**
  Пользователь запрашивает просмотр графиков данных ЭКГ, полученных от Electron приложения.

- **Вывод заключения о состоянии здоровья:**
  Пользователь запрашивает анализ состояния здоровья на основе данных ЭКГ, анализируемых искусственным интеллектом.

3.3. Разработка и реализация архитектуры.

3.3.1. Использование Docker для контейнеризации приложения.

Для упрощения процесса разработки, тестирования и развертывания серверного приложения был использован Docker. Вот основные шаги при разработке приложения с использованием Docker:

1. **Определение сервисов в файле docker-compose.yml:**
    - В файле `docker-compose.yml` определены два сервиса: `web` для серверного приложения на Flask и `postgres` для базы данных PostgreSQL.
    - Для каждого сервиса указаны настройки, такие как сборка образа, зависимости, порты, переменные окружения и т.д.
2. **Настройка среды разработки:**
    - Для сервиса `web` указаны переменные окружения `FLASK_APP` И `FLASK_ENV`, определяющие основные параметры запуска Flask приложения.
    - В качестве базового образа для серверного приложения использован официальный образ Python с поддержкой Ubuntu 22.04.
3. **Управление зависимостями и файлами приложения:**
    - В Dockerfile определены шаги для установки зависимостей из файла `requirements.txt` и копирования всех остальных файлов приложения внутрь контейнера.
    - Это позволяет изолировать приложение и его зависимости внутри контейнера, обеспечивая надежную и воспроизводимую среду выполнения.
4. **Использование внешнего образа PostgreSQL:**
    - Для сервиса `postgres` использован официальный образ PostgreSQL.
    - В volumes прописан путь к файлу `init.sql`, который будет запущен при инициализации контейнера и содержит SQL-запросы для создания базы данных и таблиц.
5. **Управление данными:**

- Для сохранения данных базы данных PostgreSQL использован Docker volume, который привязывается к директории внутри контейнера.
- Это обеспечивает сохранность данных даже при перезапуске контейнера или удалении образа.

6. **Метаданные образа и авторство:**

   - В Dockerfile добавлены метаданные, такие как метка `maintainer` с указанием контактной информации разработчика и метка `description` с кратким описанием приложения.

3.3.2. Разработка и реализация логики на Python с использованием Flask.

Для создания серверной части приложения был выбран фреймворк Flask, который написан для Python. Вот основные шаги при разработке приложения с использованием Flask:

1. **Инициализация приложения и настройка JWT:**

   - В файле app.py создается объект Flask приложения и настраивается JWT (JSON Web Tokens) для обеспечения безопасной аутентификации пользователей.
   - Для генерации токенов используется секретный ключ и указывается срок действия токена.

2. **Обработка исключений JWT:**

   - В приложении реализована обработка исключений, связанных с JWT, таких как отсутствие заголовка авторизации, неверный заголовок авторизации и истекший срок действия токена.

3. **Регистрация маршрутов:**

   - Для каждого сценария использования определены маршруты и функции-обработчики.
   - Для удобства организации и читабельности кода, маршруты разделены на модули `common`, `web` и `app` по типу сценария.

4. **Разработка функционала доступа к данным:**

- Для удобства доступа к данным и выполнения операций CRUD (создание, чтение, обновление, удаление) были реализованы модели данных и базовый класс, предоставляющий общие методы для работы с базой данных.
- Это позволяет упростить разработку и обеспечить единый подход к взаимодействию с данными из различных частей приложения.

5. **Использование библиотек:**

   - Для работы с базой данных в приложении был выбран фреймворк SQLAlchemy, обеспечивающий ORM (объектно-реляционное отображение) и удобные инструменты для работы с реляционными базами данных.
   - Это позволяет использовать объектно-ориентированный подход при работе с данными и упрощает процесс взаимодействия с базой данных.

3.3.3. Работа с базой данных.

Для обеспечения хранения и управления данными серверного приложения была выбрана реляционная база данных. Разработка и реализация базы данных включает в себя следующие шаги:

1. **Создание схемы базы данных:**

   - В ходе проектирования было определено три основных сущности: пользователи данные ЭКГ и аутентификационные данные.
   - Для каждой сущности была разработана соответствующая таблица в базе данных, представляющая собой логическую структуру для хранения данных.

2. **Определение полей и их типов:**

   - Каждая таблица содержит набор полей, определяющих характеристики сущности.
   - Типы данных для полей выбирались с учетом требований к хранению и обработке информации.

3. **Определение отношений между таблицами:**

   - Для связывания данных между таблицами были определены внешние ключи и отношения.

- Например, таблица аутентификационных данных имеет внешний ключ, связывающий ее с таблицей пользователей.

4. Заключение.

Таким образом, в результате выполнения практики были достигнуты значительные результаты в разработке серверной части приложения на основе Python Flask. Процесс разработки позволил мне ознакомиться с основными инструментами и технологиями, необходимыми для создания современных веб-приложений.

Основные достижения включают в себя:

1. Разработка и настройка маршрутов для обработки запросов от клиентских приложений на базе Electron и веб-приложений на Vue.js.
2. Реализация аутентификации и регистрации пользователей с использованием JWT для обеспечения безопасности взаимодействия с сервером.
3. Внедрение базы данных PostgreSQL с использованием SQLAlchemy для хранения данных пользователей и их аутентификационных данных.
4. Освоение Docker для удобного развертывания и управления серверным приложением в изолированных контейнерах.

5. Список литературы.

1. Гринберг, М. Flask Web Development: Developing Web Applications with Python [Текст] / М. Гринберг. - СПб.: Наука, 2018. - 300 с.
2. Гонсалес, М. PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database [Текст] / М. Гонсалес. - М.: Вильямс, 2019. - 400 с.
3. Cochrane, K. Docker Cookbook: Over 100 practical and insightful recipes to build distributed applications with Docker, 2nd Edition [Текст] / K. Cochrane. - М.: Питер, 2019. - 350 с.
4. Pallets Projects. Flask Documentation [Электронный ресурс]. - Режим доступа: https://flask.palletsprojects.com/ (дата обращения: 05.05.2024).
5. SQLAlchemy Documentation [Электронный ресурс]. - Режим доступа: https://docs.sqlalchemy.org/en/20/ (дата обращения: 05.05.2024).

6. Приложение.

Листинг 1: app.py

```
from flask import Flask, render_template
from api.app.register_routes import AppBPRegister
from api.common.register_routes import CommonBPRegister
from api.web.register_routes import WebBPRegister
from flask_jwt_extended import JWTManager
from flask_jwt_extended.exceptions import NoAuthorizationError,
    InvalidHeaderError, JWTDecodeError
from datetime import timedelta
from database.db import init_db

app = Flask(__name__)

# Register JWT
app.config['JWT_SECRET_KEY'] = 's3cr3t_k3y_f0r_jwt'
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = timedelta(days=30)
jwt = JWTManager(app)

# Handle JWT exceptions
@app.errorhandler(NoAuthorizationError)
def handle_auth_error(e):
    return jsonify({"msg": "Missing authorization header"}), 401

@app.errorhandler(InvalidHeaderError)
def handle_invalid_header_error(e):
    return jsonify({"msg": "Invalid authorization header"}), 401

@app.errorhandler(JWTDecodeError)
def handle_expired_signature_error(e):
    return jsonify({"msg": "Token has expired"}), 401

# Register blueprints
CommonBPRegister(app)
```

```
WebBPRegister(app)
AppBPRegister(app)

# Init db
app.config['SQLALCHEMY_DATABASE_URI'] =
   f'postgresql://root:root@postgres:5432/postgres'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
init_db(app)

# DONT forget to remove
@app.route('/')
def welcome():
    return render_template('welcome.html')

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Листинг 2: database/db.py

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def init_db(app):
    db.init_app(app)

class DBStatus:
    OK = "OK"
    ERROR = "ERROR"

"""Base model class providing common database operations"""
class BaseModel:

    # Create a new record in the database
    def create(self):
        try:
            db.session.add(self)
```

```
            db.session.commit()
            return DBStatus.OK
        except Exception as e:
            db.session.rollback()
            print(f"Error creating record: {e}")
            return DBStatus.ERROR

    # Find a record by its ID
    @classmethod
    def find_by_id(cls, id):
        try:
            return cls.query.filter_by(id=id).first(), DBStatus.OK
        except Exception as e:
            print(f"Error finding record by ID: {e}")
            return None, DBStatus.ERROR

    # Update an existing record
    def update(self, **kwargs):
        try:
            for key, value in kwargs.items():
                setattr(self, key, value)
            db.session.commit()
            return DBStatus.OK
        except Exception as e:
            db.session.rollback()
            print(f"Error updating record: {e}")
            return DBStatus.ERROR

    # Delete an existing record
    def delete(self):
        try:
            db.session.delete(self)
            db.session.commit()
            return DBStatus.OK
        except Exception as e:
            db.session.rollback()
```

```
            print(f"Error deleting record: {e}")
            return DBStatus.ERROR

    # Return all records of this model
    @classmethod
    def return_all(cls):
        try:
            return cls.query.all(), DBStatus.OK
        except Exception as e:
            print(f"Error retrieving all records: {e}")
            return None, DBStatus.ERROR

"""Model representing users in the database."""
class users(db.Model, BaseModel):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    full_name = db.Column(db.String(100))
    gender = db.Column(db.String(10))
    age = db.Column(db.Integer)
    weight = db.Column(db.Float)
    date_of_birth = db.Column(db.Date)

"""Model representing records in the database."""
class records(db.Model, BaseModel):
    __tablename__ = 'records'

    id = db.Column(db.Integer, primary_key=True)
    reading_date = db.Column(db.Date, nullable=False)
    time_interval = db.Column(db.Interval, nullable=False)
    readings_data = db.Column(db.JSON)

"""Model representing authentication credentials in the database."""
class auth(db.Model, BaseModel):
    __tablename__ = 'auth'
```

```
    id = db.Column(db.Integer, primary_key=True)
    login = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'),
       nullable=False)

    user = db.relationship('users', backref=db.backref('auth',
       lazy=True))

    # Find an authentication record by login
    @classmethod
    def find_by_login(cls, login):
        try:
            return cls.query.filter_by(login=login).first(), DBStatus.OK
        except Exception as e:
            print(f"Error finding record by login: {e}")
            return None, DBStatus.ERROR

    # Find an authentication record by email
    @classmethod
    def find_by_email(cls, email):
        try:
            return cls.query.filter_by(email=email).first(), DBStatus.OK
        except Exception as e:
            print(f"Error finding record by email: {e}")
            return None, DBStatus.ERROR
```

Листинг 3: test/basic_post.py

```
import requests

url = 'http://127.0.0.1:5000/api/register'

data = {
    'login': 'va',
    'password': 'bd',
```

```
    'email': "bch09@rambler.com"
}

response = requests.post(url, json=data)

if response.status_code ==  200:
    print('POST request was successful.')
    print('Content\n')
    print(response.json())
else:
    print(f'POST request failed with status code
       {response.status_code}.')

print(response.text)
```

Листинг 4: api/web/register_routes.py

```
from api.web.users.routes import web_users_bp

common_prefix='/api/web'

class WebBPRegister:
    def __init__(self, app):
        self.app = app
        self.register_blueprints()

    def register_blueprints(self):
        self.app.register_blueprint(web_users_bp,
           url_prefix=common_prefix)
```

Листинг 5: api/web/users/routes.py

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required, get_jwt_identity
from database.db import Users

web_bp = Blueprint('web', __name__)
```

```python
@web_bp.route('/update_profile', methods=['POST'])
def update_profile():
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    # Get data from the request
    login = request.json.get('login')
    birthdate = request.json.get('birthdate')

    if not login or not birthdate:
        return jsonify({"msg": "Missing login or birthdate"}), 400

    # Update user profile in the database
    user = Users.query.filter_by(login=login).first()

    if not user:
        return jsonify({"msg": "User not found"}), 404

    user.birthdate = birthdate

    # Commit changes to the database
    db.session.commit()

    return jsonify({"msg": "Profile updated successfully"}), 200

@web_bp.route('/view_ecg', methods=['GET'])
def view_ecg():
    # Get the current user from JWT
    current_user = get_jwt_identity()

    # Get ECG data from the database
    ecg_data = ecg_data.get_user_ecg_data(current_user)

    if not ecg_data:
        return jsonify({"msg": "No ECG data found"}), 404
```

```
    # Send ECG data in the response
    return jsonify({"ecg_data": ecg_data}), 200

@web_bp.route('/health_analysis', methods=['GET'])
def health_analysis():
    # Get the current user from JWT
    current_user = get_jwt_identity()

    # Get ECG data from the database
    ecg_data = ecg_data.get_user_ecg_data(current_user)

    if not ecg_data:
        return jsonify({"msg": "No ECG data found"}), 404

    # Analyze ECG data using artificial intelligence
    health_result = ai_analysis.analyze_ecg_data(ecg_data)

    # Send health analysis result in the response
    return jsonify({"health_analysis": health_result}), 200
```

Листинг 6: api/common/register_routes.py

```
from api.common.auth.routes import common_auth_bp

common_prefix='/api'

class CommonBPRegister:
    def __init__(self, app):
        self.app = app
        self.register_blueprints()

    def register_blueprints(self):
        self.app.register_blueprint(common_auth_bp,
           url_prefix=common_prefix)
```

Листинг 7: api/common/auth/routes.py

```
from flask import Blueprint, request, jsonify
```

```python
from flask_jwt_extended import create_access_token,
   create_refresh_token, jwt_required
from database.db import auth, DBStatus, users

common_auth_bp = Blueprint('auth', __name__)

@common_auth_bp.route('/login', methods=['POST'])
def login_post():
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    login = request.json.get('login')
    password = request.json.get('password')

    if not login or not password:
        return jsonify({"msg": "Missing login or password"}), 400

    auth_record, status = auth.find_by_login(login)

    if status == DBStatus.ERROR or auth_record is None:
        return jsonify({"msg": "Invalid login credentials"}), 401

    if auth_record.password != password:
        return jsonify({"msg": "Invalid login credentials"}), 401

    access_token = create_access_token(identity=login)
    refresh_token = create_refresh_token(identity=login)
    return jsonify({"msg": "Login successful", "access_token":
       access_token, "refresh_token": refresh_token}), 200

@common_auth_bp.route('/refresh', methods=['POST'])
@jwt_required(refresh=True)
def refresh():
    current_user = get_jwt_identity()
    access_token = create_access_token(identity=current_user)
    return jsonify({"access_token": access_token}), 200
```

```python
@common_auth_bp.route('/register', methods=['POST'])
def register_post():
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    login = request.json.get('login')
    password = request.json.get('password')
    email = request.json.get('email')

    if not login or not password or not email:
        return jsonify({"msg": "Missing required fields"}), 400

    existing_auth_record, status = auth.find_by_login(login)

    if status == DBStatus.OK and existing_auth_record:
        return jsonify({"msg": "Login already exists"}), 400

    existing_email_record, status = auth.find_by_email(email)

    if status == DBStatus.OK and existing_email_record:
        return jsonify({"msg": "Email already registered"}), 400

    new_user_record = users()
    status_user = new_user_record.create()

    if status_user != DBStatus.OK:
        return jsonify({"msg": "Failed to register user"}), 500

    new_auth_record = auth(login=login, password=password, email=email,
       user_id=new_user_record.id)
    status = new_auth_record.create()

    if status == DBStatus.OK:
        return jsonify({"msg": "Registration successful"}), 200
    else:
```

```
        return jsonify({"msg": "Failed to register user"}), 500

""" REMOVE """
@common_auth_bp.route('/auths', methods=['GET'])
def get_all_auths():
    from database.db import auth
    auths, status = auth.return_all()

    if status == DBStatus.OK:
        user_data = [{"id": auth.id, "login": auth.login, "email":
            auth.email, "id_user": auth.user_id} for auth in auths]
        return jsonify({"auths": user_data}), 200
    else:
        return jsonify({"msg": "Failed to fetch users"}), 500

""" REMOVE """
@common_auth_bp.route('/users', methods=['GET'])
def get_all_users():
    from database.db import users
    users, status = users.return_all()

    if status == DBStatus.OK:
        user_data = [{"id": users.id} for users in users]
        return jsonify({"users": user_data}), 200
    else:
        return jsonify({"msg": "Failed to fetch users"}), 500
```

Листинг 8: api/app/register_routes.py

```
from api.common.auth.routes import app_auth_bp

common_prefix='/api/app'

class AppBPRegister:
    def __init__(self, app):
        self.app = app
        self.register_blueprints()
```

```
    def register_blueprints(self):
        self.app.register_blueprint(app_auth_bp,
           url_prefix=common_prefix)
```

Листинг 9: api/app/routes.py

```
from flask import Blueprint, request, jsonify
from database.db import ecg_data, ai_analysis

electron_bp = Blueprint('electron', __name__)

@electron_bp.route('/upload_ecg', methods=['POST'])
def upload_ecg():
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    # Get ECG data from the request
    ecg_data = request.json.get('ecg_data')

    if not ecg_data:
        return jsonify({"msg": "Missing ECG data"}), 400

    # Save ECG data to the database
    new_ecg_record = ecg_data(ecg_data=ecg_data)
    new_ecg_record.save()

    return jsonify({"msg": "ECG data uploaded successfully"}), 200

@electron_bp.route('/analyze_ecg', methods=['POST'])
def analyze_ecg():
    if not request.is_json:
        return jsonify({"msg": "Missing JSON in request"}), 400

    # Get ECG data from the request
    ecg_data = request.json.get('ecg_data')
```

```
    if not ecg_data:
        return jsonify({"msg": "Missing ECG data"}), 400

    # Analyze ECG data using artificial intelligence
    analysis_result = ai_analysis.analyze_ecg(ecg_data)

    return jsonify({"analysis_result": analysis_result}), 200
```

Листинг 10: Dockerfile

```
# Use the official Python image with Ubuntu 22.04 support
FROM python:3.9

# Set environment variable to avoid buffering issues
ENV PYTHONUNBUFFERED 1

# Set the working directory inside the container
WORKDIR /app

# Copy and install dependencies
COPY requirements.txt requirements.txt
RUN apt-get update
RUN pip install --no-cache-dir -r requirements.txt

# Copy all other files into the container
COPY . .

# Add necessary metadata
LABEL maintainer="Daniil Bulgakov <d.bulgakov@gmail.com>"
LABEL description="Basic flask back-end app for ECG ESP32"

# Run the Flask application using the built-in development server
CMD ["flask", "run", "--host=0.0.0.0"]
```

Листинг 11: docker-compose.yml

```
version: '3.8'
```

```
services:
 web:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - postgres
    ports:
      - "5000:5000"
    environment:
      - FLASK_APP=app.py
      - FLASK_ENV=development
    volumes:
      - .:/app
 postgres:
    image: postgres
    restart: always
    ports:
      - "5432:5432"
    volumes:
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
      - postgres_data:/var/lib/postgresql/data
    command: ["postgres", "-c", "log_statement=all"]

volumes:
 postgres_data:
```