

Blood & Fitness App - Backend Code Documentation

1. Main Backend Server (API & Logic)

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from flask_sqlalchemy import SQLAlchemy
import bcrypt
import jwt
import os
import werkzeug
from datetime import datetime, timedelta

# Initialize Flask App
app = Flask(__name__)
CORS(app)

# Configuration
SECRET_KEY = os.getenv('JWT_SECRET_KEY', 'your-secret-key-change-in-production')
UPLOAD_DIR = "uploads"
if not os.path.exists(UPLOAD_DIR):
    os.makedirs(UPLOAD_DIR)

# Database Configuration
# Use SQLite locally, but allow overriding with DATABASE_URL for PostgreSQL on Render/Neon
db_path = os.path.join(os.path.abspath(os.path.dirname(__file__)), 'bloodfit.db')
app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv('DATABASE_URL', f'sqlite:/// {db_path}')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# =====
# DATABASE MODELS
# =====

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    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)
    profile = db.relationship('Profile', backref='user', uselist=False)

class Profile(db.Model):
    __tablename__ = 'profiles'
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    name = db.Column(db.String(100))
    age = db.Column(db.Integer)
    gender = db.Column(db.String(20))
    height = db.Column(db.Integer)
    heightCm = db.Column(db.Integer)
    weight = db.Column(db.Float)
    blood_group = db.Column(db.String(10))
    diseases = db.Column(db.Text)
    allergies = db.Column(db.Text)
    notes = db.Column(db.Text)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    def to_dict(self):
        return {
            'id': self.id,
            'user_id': self.user_id,
```

Blood & Fitness App - Backend Code Documentation

```
'name': self.name,
'age': self.age,
'gender': self.gender,
'height': self.height,
'heightCm': self.heightCm,
'weight': self.weight,
'bloodGroup': self.blood_group, # Map back to frontend expectation
'diseases': self.diseases,
'allergies': self.allergies,
'notes': self.notes,
'updated_at': self.updated_at.isoformat() if self.updated_at else None
}

# Create Tables
with app.app_context():
    db.create_all()

@app.route('/', methods=['GET'])
def index():
    return jsonify({
        'status': 'ok',
        'message': 'Blood & Fit API is online (SQLAlchemy Mode)'
    }), 200

# =====
# HEALTH CHECK
# =====

@app.route('/health', methods=['GET'])
def health_check():
    return jsonify({
        'status': 'ok',
        'message': 'Blood & Fit API is running',
        'database': 'connected'
    }), 200

# =====
# AUTHENTICATION
# =====

@app.route('/api/register', methods=['POST'])
def register():
    try:
        data = request.json
        email = data.get('email')
        password = data.get('password')

        if not email or not password:
            return jsonify({'error': 'Email and password are required'}), 400

        # Check if user exists
        if User.query.filter_by(email=email).first():
            return jsonify({'error': 'Email already exists'}), 409

        # Hash password
        hashed_pw = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

        # Create user
        new_user = User(email=email, password_hash=hashed_pw)
        db.session.add(new_user)
        db.session.commit()
```

Blood & Fitness App - Backend Code Documentation

```
# Create default profile
new_profile = Profile(user_id=new_user.id, name=email.split('@')[0])
db.session.add(new_profile)
db.session.commit()

# Generate Token
token = jwt.encode({
    'user_id': new_user.id,
    'email': new_user.email,
    'exp': datetime.utcnow() + timedelta(days=7)
}, SECRET_KEY, algorithm='HS256')

return jsonify({
    'success': True,
    'token': token,
    'user': {'email': email, 'id': new_user.id}
}), 201

except Exception as e:
    db.session.rollback()
    print(f"Registration error: {str(e)}")
    return jsonify({'error': 'Registration failed'}), 500

@app.route('/api/login', methods=['POST'])
def login():
    try:
        data = request.json
        email = data.get('email')
        password = data.get('password')

        user = User.query.filter_by(email=email).first()

        if not user or not bcrypt.checkpw(password.encode('utf-8'), user.password_hash.encode('utf-8')):
            return jsonify({'error': 'Invalid email or password'}), 401

        token = jwt.encode({
            'user_id': user.id,
            'email': user.email,
            'exp': datetime.utcnow() + timedelta(days=7)
        }, SECRET_KEY, algorithm='HS256')

        return jsonify({
            'success': True,
            'token': token,
            'user': {'email': user.email, 'id': user.id}
        }), 200

    except Exception as e:
        print(f"Login error: {str(e)}")
        return jsonify({'error': 'Login failed'}), 500

@app.route('/api/profile', methods=['GET'])
def get_profile():
    try:
        auth_header = request.headers.get('Authorization')
        if not auth_header or not auth_header.startswith('Bearer '):
            return jsonify({'error': 'No token provided'}), 401

        token = auth_header.split(' ')[1]
        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
            user_id = payload['user_id']
```

Blood & Fitness App - Backend Code Documentation

```
except jwt.InvalidTokenError:
    return jsonify({'error': 'Invalid token'}), 401

profile = Profile.query.filter_by(user_id=user_id).first()

if not profile:
    return jsonify({'error': 'Profile not found'}), 404

return jsonify({
    'success': True,
    'profile': profile.to_dict()
}), 200

except Exception as e:
    print(f"Profile error: {str(e)}")
    return jsonify({'error': 'Failed to fetch profile'}), 500

@app.route('/api/profile', methods=['PUT'])
def update_profile():
    try:
        auth_header = request.headers.get('Authorization')
        if not auth_header or not auth_header.startswith('Bearer '):
            return jsonify({'error': 'No token provided'}), 401

        token = auth_header.split(' ')[1]
        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
            user_id = payload['user_id']
        except jwt.InvalidTokenError:
            return jsonify({'error': 'Invalid token'}), 401

        data = request.json
        profile = Profile.query.filter_by(user_id=user_id).first()

        if not profile:
            return jsonify({'error': 'Profile not found'}), 404

        # Update fields
        profile.name = data.get('name', profile.name)
        profile.age = data.get('age', profile.age)
        profile.gender = data.get('gender', profile.gender)
        profile.height = data.get('height', profile.height)
        profile.heightCm = data.get('heightCm', profile.heightCm)
        profile.weight = data.get('weight', profile.weight)
        profile.blood_group = data.get('bloodGroup', profile.blood_group)
        profile.diseases = data.get('diseases', profile.diseases)
        profile.allergies = data.get('allergies', profile.allergies)
        profile.notes = data.get('notes', profile.notes)
        profile.updated_at = datetime.utcnow()

        db.session.commit()
        return jsonify({'success': True, 'message': 'Profile updated'}), 200

    except Exception as e:
        db.session.rollback()
        print(f"Update error: {str(e)}")
        return jsonify({'error': 'Failed to update profile'}), 500

# =====
# ML CONFIGURATION & LOADING
# =====
```

Blood & Fitness App - Backend Code Documentation

```
import joblib
import pandas as pd
import numpy as np

# Path to the trained model
MODEL_PATH = "ml_models_advanced/best_blood_model.pkl"
ML_ARTIFACTS = None

# Load Model on Startup
try:
    if os.path.exists(MODEL_PATH):
        print(f"? Loading Advanced ML Model from {MODEL_PATH}...")
        ML_ARTIFACTS = joblib.load(MODEL_PATH)
        print("? Model & Artifacts Loaded Successfully")
    else:
        print(f"? Model file not found at {MODEL_PATH}. Prediction endpoint will return 503.")
except Exception as e:
    print(f"? Failed to load model: {e}")

# =====
# ML ENDPOINTS
# =====

@app.route('/api/predict-disease', methods=['POST'])
def predict_disease():
    if not ML_ARTIFACTS:
        return jsonify({"error": "ML Model not loaded on server"}), 503

    try:
        data = request.json
        print(f"? Received Prediction Request: {data}")

        # 1. Prepare Dataframe with correct column order
        feature_names = ML_ARTIFACTS['feature_names']

        # Create input dictionary - careful handling of missing values
        input_data = {}
        for col in feature_names:
            val = data.get(col, 0.0)
            try:
                val = float(val)
            except:
                val = 0.0
            input_data[col] = [val]

        df_input = pd.DataFrame(input_data)

        # 2. Preprocessing: Scale Features
        scaler = ML_ARTIFACTS['scaler']
        X_scaled = pd.DataFrame(scaler.transform(df_input[feature_names]), columns=feature_names)

        # 3. Predict
        model = ML_ARTIFACTS['model']
        pred_idx = model.predict(X_scaled)[0]
        # Decode the label (0 -> "Healthy", 1 -> "Anemia", etc.)
        pred_label = ML_ARTIFACTS['le_target'].inverse_transform([pred_idx])[0]

        # Get Probability/Confidence
        confidence = 0.0
        if hasattr(model, "predict_proba"):
            probs = model.predict_proba(X_scaled)
            confidence = float(np.max(probs))
```

Blood & Fitness App - Backend Code Documentation

```
print(f"? Prediction: {pred_label} ({confidence:.2f})")

return jsonify({
    "status": "success",
    "prediction": pred_label,
    "confidence": f"{confidence*100:.1f}%",
    "message": "Analysis based on random forest model"
})

except Exception as e:
    print(f"? Prediction Error: {e}")
    return jsonify({"error": str(e)}), 400

@app.route('/analyze', methods=['POST'])
def analyze_report():
    from extract import extract_data
    if 'file' not in request.files:
        return jsonify({"error": "No file part"}), 400

    file = request.files['file']
    if file.filename == '':
        return jsonify({"error": "No selected file"}), 400

    if file:
        filename = werkzeug.utils.secure_filename(file.filename)
        filepath = os.path.join(UPLOAD_DIR, filename)
        file.save(filepath)

        try:
            print(f"? Processing {filename} with OCR and Extraction...")
            results = extract_data(filepath)

            return jsonify({
                "message": "Analysis Complete",
                "raw_results": results,
                "ml_enabled": True
            })

        except Exception as e:
            print(f"? Error during extraction: {e}")
            return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    port = int(os.getenv('PORT', 5000))
    print(f"? Blood & Fit Unified API (SQLAlchemy + ML) running on port {port}")
    app.run(host='0.0.0.0', port=port, debug=True)
```

Blood & Fitness App - Backend Code Documentation

2. Database Models & Schema

```
import sqlite3
import os
from datetime import datetime

DATABASE_PATH = os.getenv('DATABASE_PATH', 'bloodfit.db')

def init_db():
    """Initialize the database with required tables"""
    conn = sqlite3.connect(DATABASE_PATH)
    cursor = conn.cursor()

    # Users table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            email TEXT UNIQUE NOT NULL,
            password_hash TEXT NOT NULL,
            created_at TEXT NOT NULL
        )
    ''')

    # Profiles table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS profiles (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            name TEXT,
            age INTEGER,
            gender TEXT,
            height INTEGER,
            heightCm INTEGER,
            weight REAL,
            blood_group TEXT,
            diseases TEXT,
            allergies TEXT,
            notes TEXT,
            updated_at TEXT,
            FOREIGN KEY (user_id) REFERENCES users (id)
        )
    ''')

    conn.commit()
    conn.close()
    print("Database initialized successfully")

def get_db_connection():
    """Get a database connection"""
    conn = sqlite3.connect(DATABASE_PATH)
    conn.row_factory = sqlite3.Row
    return conn

if __name__ == '__main__':
    init_db()
```

Blood & Fitness App - Backend Code Documentation

3. Authentication Logic

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import bcrypt
import jwt
import os
from datetime import datetime, timedelta
from database import init_db, get_db_connection

app = Flask(__name__)
CORS(app) # Enable CORS for frontend

# Secret key for JWT (use environment variable in production)
SECRET_KEY = os.getenv('JWT_SECRET_KEY', 'your-secret-key-change-in-production')

# Initialize database on startup
init_db()

@app.route('/api/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({'status': 'ok', 'message': 'Auth API is running'})

@app.route('/api/register', methods=['POST'])
def register():
    """Register a new user"""
    try:
        data = request.json
        email = data.get('email')
        password = data.get('password')

        if not email or not password:
            return jsonify({'error': 'Email and password are required'}), 400

        # Hash password
        password_hash = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        # Insert into database
        conn = get_db_connection()
        cursor = conn.cursor()

        try:
            cursor.execute(
                'INSERT INTO users (email, password_hash, created_at) VALUES (?, ?, ?)',
                (email, password_hash.decode('utf-8'), datetime.utcnow().isoformat())
            )
            user_id = cursor.lastrowid

            # Create default profile
            cursor.execute(
                'INSERT INTO profiles (user_id, name, updated_at) VALUES (?, ?, ?)',
                (user_id, email.split('@')[0], datetime.utcnow().isoformat())
            )

            conn.commit()

            # Generate JWT token
            token = jwt.encode({
                'user_id': user_id,
                'email': email,
```

Blood & Fitness App - Backend Code Documentation

```
        'exp': datetime.utcnow() + timedelta(days=7)
    }, SECRET_KEY, algorithm='HS256')

    return jsonify({
        'success': True,
        'token': token,
        'user': {'email': email, 'id': user_id}
    }), 201

except Exception as e:
    conn.rollback()
    if 'UNIQUE constraint failed' in str(e):
        return jsonify({'error': 'Email already exists'}), 409
    raise
finally:
    conn.close()

except Exception as e:
    print(f"Registration error: {str(e)}")
    return jsonify({'error': 'Registration failed'}), 500

@app.route('/api/login', methods=['POST'])
def login():
    """Login user"""
    try:
        data = request.json
        email = data.get('email')
        password = data.get('password')

        if not email or not password:
            return jsonify({'error': 'Email and password are required'}), 400

        # Get user from database
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users WHERE email = ?', (email,))
        user = cursor.fetchone()
        conn.close()

        if not user:
            return jsonify({'error': 'Invalid email or password'}), 401

        # Check password
        if not bcrypt.checkpw(password.encode('utf-8'), user['password_hash'].encode('utf-8')):
            return jsonify({'error': 'Invalid email or password'}), 401

        # Generate JWT token
        token = jwt.encode({
            'user_id': user['id'],
            'email': user['email'],
            'exp': datetime.utcnow() + timedelta(days=7)
        }, SECRET_KEY, algorithm='HS256')

        return jsonify({
            'success': True,
            'token': token,
            'user': {'email': user['email'], 'id': user['id']}
        }), 200

    except Exception as e:
        print(f"Login error: {str(e)}")
        return jsonify({'error': 'Login failed'}), 500
```

Blood & Fitness App - Backend Code Documentation

```
@app.route('/api/profile', methods=['GET'])
def get_profile():
    """Get user profile"""
    try:
        # Get token from header
        auth_header = request.headers.get('Authorization')
        if not auth_header or not auth_header.startswith('Bearer '):
            return jsonify({'error': 'No token provided'}), 401

        token = auth_header.split(' ')[1]

        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
            user_id = payload['user_id']
        except jwt.ExpiredSignatureError:
            return jsonify({'error': 'Token expired'}), 401
        except jwt.InvalidTokenError:
            return jsonify({'error': 'Invalid token'}), 401

        # Get profile from database
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM profiles WHERE user_id = ?', (user_id,))
        profile = cursor.fetchone()
        conn.close()

        if not profile:
            return jsonify({'error': 'Profile not found'}), 404

        return jsonify({
            'success': True,
            'profile': dict(profile)
        }), 200

    except Exception as e:
        print(f"Profile fetch error: {str(e)}")
        return jsonify({'error': 'Failed to fetch profile'}), 500

@app.route('/api/profile', methods=['PUT'])
def update_profile():
    """Update user profile"""
    try:
        # Get token from header
        auth_header = request.headers.get('Authorization')
        if not auth_header or not auth_header.startswith('Bearer '):
            return jsonify({'error': 'No token provided'}), 401

        token = auth_header.split(' ')[1]

        try:
            payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
            user_id = payload['user_id']
        except jwt.ExpiredSignatureError:
            return jsonify({'error': 'Token expired'}), 401
        except jwt.InvalidTokenError:
            return jsonify({'error': 'Invalid token'}), 401

        data = request.json

        # Update profile in database
        conn = get_db_connection()
```

Blood & Fitness App - Backend Code Documentation

```
cursor = conn.cursor()

cursor.execute('''
    UPDATE profiles SET
        name = ?, age = ?, gender = ?, height = ?, heightCm = ?,
        weight = ?, blood_group = ?, diseases = ?, allergies = ?,
        notes = ?, updated_at = ?
    WHERE user_id = ?
''', (
    data.get('name'), data.get('age'), data.get('gender'),
    data.get('height'), data.get('heightCm'), data.get('weight'),
    data.get('bloodGroup'), data.get('diseases'), data.get('allergies'),
    data.get('notes'), datetime.utcnow().isoformat(), user_id
))

conn.commit()
conn.close()

return jsonify({'success': True, 'message': 'Profile updated'}), 200

except Exception as e:
    print(f"Profile update error: {str(e)}")
    return jsonify({'error': 'Failed to update profile'}), 500

if __name__ == '__main__':
    port = int(os.getenv('PORT', 5000))
    app.run(host='0.0.0.0', port=port, debug=True)
```

Blood & Fitness App - Backend Code Documentation

4. ML Model Simulation Script

```
import time
import random

def print_header(title):
    print("\n" + "="*50)
    print(f"    ? ML MODEL DEMO: {title}")
    print("="*50 + "\n")

def simulate_delay(seconds, message):
    print(f"? {message}...", end="", flush=True)
    time.sleep(seconds)
    print(" ? Done!")

def show_dataset():
    print_header("Loading Training Dataset")
    simulate_delay(1.0, "Reading CSV from disk")

    print("\n? Dataset Preview (First 5 Rows):")
    data = [
        {"id": 1, "hemoglobin": 11.2, "wbc": 4500, "label": "Anemia"},
        {"id": 2, "hemoglobin": 14.5, "wbc": 7200, "label": "Healthy"},
        {"id": 3, "hemoglobin": 10.1, "wbc": 12000, "label": "Infection"},
        {"id": 4, "hemoglobin": 15.0, "wbc": 6800, "label": "Healthy"},
        {"id": 5, "hemoglobin": 13.0, "wbc": 15000, "label": "Leukocytosis"}
    ]

    print(f"{'ID':<5} | {'Hemoglobin':<12} | {'WBC':<8} | {'Label'}")
    print("-" * 50)
    for r in data:
        print(f"{r['id']:<5} | {r['hemoglobin']:<12} | {r['wbc']:<8} | {r['label']}")
    print("-" * 50)
    print("\nTotal Samples: 10,000")

def simulate_training():
    print_header("Training RandomForest Classifier")

    print("? Initializing model parameters: n_estimators=100, max_depth=5")
    simulate_delay(0.5, "Splitting train/test data (80/20)")

    print("\n?? Training Progress:")
    for i in range(1, 6):
        print(f"Epoch {i}/5: Loss: {random.uniform(0.1, 0.5):.4f} - Accuracy: {80 + i*3:.2f}%")
        time.sleep(0.5)

    print("\n? Saving model weights to 'model.pkl'")
    simulate_delay(1.0, "Serializing model")

    print("\n? Final Model Accuracy: 96.5%")

def make_prediction():
    print_header("Live Prediction Test")

    h = float(input("Enter Hemoglobin Level: "))
    w = int(input("Enter WBC Count: "))

    print(f"\n? Input Tensor: [{h}, {w}]")

    simulate_delay(0.8, "Loading 'model.pkl'")
    simulate_delay(0.5, "Running inference")
```

Blood & Fitness App - Backend Code Documentation

```
# Simple logic for demo
prediction = "Healthy"
if h < 12: prediction = "Anemia"
elif w > 11000: prediction = "Infection"

confidence = random.uniform(0.85, 0.99)

print(f"\n? PREDICTION RESULT: {prediction}")
print(f"? Confidence Score: {confidence:.2%}")

def main():
    while True:
        print("\n? MACHINE LEARNING PIPELINE:")
        print("1. View Dataset (Inspect Data)")
        print("2. Train Model (Simulate Training)")
        print("3. Test Prediction (Inference)")
        print("4. Exit")

        choice = input("\nSelect operation (1-4): ")

        if choice == '1':
            show_dataset()
            input("\nPress Enter to continue...")
        elif choice == '2':
            simulate_training()
            input("\nPress Enter to continue...")
        elif choice == '3':
            make_prediction()
            input("\nPress Enter to continue...")
        elif choice == '4':
            break
        else:
            print("? Invalid choice")

if __name__ == "__main__":
    main()
```

Blood & Fitness App - Backend Code Documentation

5. API Simulation Script

```
import time
import json
import random

def print_header(title):
    print("\n" + "="*50)
    print(f" ? BACKEND API SERVER DEMO: {title}")
    print("="*50 + "\n")

def simulate_delay(seconds, message):
    print(f"? {message}...", end="", flush=True)
    time.sleep(seconds)
    print(" ? Done!")

def api_endpoint_predict_disease(blood_data):
    print_header("/api/predict-disease (POST)")

    print("? Incoming Request Payload:")
    print(json.dumps(blood_data, indent=4))

    simulate_delay(1.0, "Validating access token")
    simulate_delay(0.5, "Parsing JSON data")

    # Simulate processing
    hemoglobin = blood_data.get('hemoglobin', 0)
    wbc = blood_data.get('wbc', 0)

    print(f"\n? Analyzing Blood Parameters:")
    print(f" - Hemoglobin: {hemoglobin} g/dL")
    print(f" - WBC Count: {wbc} /mCL")

    simulate_delay(1.5, "Running Machine Learning Model (RandomForest)")

    # Mock logic
    result = "Healthy"
    if hemoglobin < 12:
        result = "Anemia Detected"
    elif wbc > 11000:
        result = "Infection / Leukocytosis"

    response = {
        "status": "success",
        "prediction": result,
        "confidence": f"{random.randint(85, 99)}%",
        "recommendations": [
            "Consult a general physician",
            "Maintain a balanced diet rich in iron"
        ]
    }

    print("\n? Sending JSON Response:")
    print(json.dumps(response, indent=4))

    return response

def api_endpoint_login():
    print_header("/api/login (POST)")

    email = input("Enter simulated email: ")
```

Blood & Fitness App - Backend Code Documentation

```
password = input("Enter simulated password: ")

payload = {"email": email, "password": "*" * len(password)}
print("\n? Incoming Request Payload:")
print(json.dumps(payload, indent=4))

simulate_delay(0.8, "Hashing password")
simulate_delay(0.8, "Checking database for user")

token = f"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.{{random.randint(10000,99999)}}"

response = {
    "status": "success",
    "token": token,
    "message": "Login successful"
}

print("\n? Sending JSON Response:")
print(json.dumps(response, indent=4))

def main():
    while True:
        print("\n? API SERVER ENDPOINTS:")
        print("1. Test Login API (/api/login)")
        print("2. Test Disease Prediction API (/api/predict-disease)")
        print("3. Exit")

        choice = input("\nSelect an endpoint to test (1-3): ")

        if choice == '1':
            api_endpoint_login()
            input("\nPress Enter to continue...")
        elif choice == '2':
            sample_data = {
                "hemoglobin": float(input("Enter Hemoglobin (e.g., 13.5): ")),
                "wbc": int(input("Enter WBC Count (e.g., 7000): ")),
                "platelets": 250000,
                "rbc": 4.8
            }
            api_endpoint_predict_disease(sample_data)
            input("\nPress Enter to continue...")
        elif choice == '3':
            break
        else:
            print("? Invalid choice")

if __name__ == "__main__":
    main()
```

Blood & Fitness App - Backend Code Documentation

6. Database Simulation Script

```
import time
import random
import uuid
import datetime

def print_header(title):
    print("\n" + "="*50)
    print(f" ? DATABASE OPERATIONS DEMO: {title}")
    print("="*50 + "\n")

def simulate_delay(seconds, message):
    print(f"? {message}...", end="", flush=True)
    time.sleep(seconds)
    print(" ? Done!")

def create_user_collection():
    print_header("Initializing Firestore DB")
    simulate_delay(1.0, "Connecting to Firebase Project: blood-fitness-app")
    simulate_delay(0.5, "Authenticating service account")
    print("\n? Database Connection Established: firestore://databases/(default)/documents")

def insert_new_user():
    print_header("CREATE: Insert New User")

    name = input("Enter user name: ")
    email = input("Enter email: ")
    user_id = str(uuid.uuid4())[:8]

    user_doc = {
        "userId": user_id,
        "name": name,
        "email": email,
        "created_at": datetime.datetime.now().isoformat(),
        "role": "patient"
    }

    print("\n? Preparing Document:")
    print(user_doc)

    simulate_delay(0.8, f"Writing to collection 'users/{user_id}'")

    print("\n? Record Inserted Successfully!")
    return user_id

def query_blood_reports(user_id):
    print_header("READ: Query Blood Reports")

    simulate_delay(0.5, f"Querying collection 'blood_reports' where userId == '{user_id}'")

    # Simulate data fetching
    reports = [
        {"date": "2023-01-15", "hemoglobin": 12.5, "status": "Normal"},
        {"date": "2023-06-20", "hemoglobin": 13.2, "status": "Normal"},
        {"date": "2024-02-10", "hemoglobin": 11.8, "status": "Low Iron"}
    ]

    print(f"\nFound {len(reports)} records for user {user_id}:")
    print("-" * 60)
    print(f"{'Date':<15} | {'Hemoglobin':<12} | {'Status'}")
```

Blood & Fitness App - Backend Code Documentation

```
print("-" * 60)

for r in reports:
    print(f"{r['date']:<15} | {r['hemoglobin']:<12} | {r['status']}")
print("-" * 60)

def main():
    create_user_collection()

    while True:
        print("\n? DATABASE OPERATIONS:")
        print("1. Register New User (CREATE)")
        print("2. Retrieve Reports (READ)")
        print("3. Exit")

        choice = input("\nSelect operation (1-3): ")

        if choice == '1':
            latest_user_id = insert_new_user()
            input("\nPress Enter to continue...")
        elif choice == '2':
            uid = input("Enter User ID (or press Enter for demo ID): ") or "user_12345"
            query_blood_reports(uid)
            input("\nPress Enter to continue...")
        elif choice == '3':
            break
        else:
            print("? Invalid choice")

if __name__ == "__main__":
    main()
```

Blood & Fitness App - Backend Code Documentation

7. Backend Requirements

```
flask==3.0.0
flask-cors==4.0.0
flask-sqlalchemy==3.1.1
bcrypt==4.1.2
pyjwt==2.8.0
psycopg2-binary==2.9.9
gunicorn==21.2.0
numpy==1.26.2
pandas==2.1.3
scikit-learn==1.3.2
joblib==1.3.2
pillow==10.1.0
pytesseract==0.3.10
werkzeug==3.0.1
```