**MAIN FILE**

from fastapi import FastAPI, Request

from fastapi.middleware.cors import CORSMiddleware

from fastapi.staticfiles import StaticFiles

from fastapi.responses import JSONResponse

from fastapi.exception\_handlers import RequestValidationError

from predict import predict\_router

from settings import user\_router, admin\_router as settings\_admin\_router

from adminlogs import admin\_router as adminlogs\_router

from negotiation import router as negotiation\_router

from investment import investment\_router

from loan\_simulator import loan\_router

from markettrends import router as markettrends\_router

from expencetracker import router as expense\_router

from notification import router as notification\_router

from rewards import router as rewards\_router

from feedback import router as feedback\_router

from systemlogs import log\_router

from auth import router as auth\_router

from chatbot import router as chatbot\_router

from biometric import biometric\_router

from emergencyloan import router as emergency\_router

from loanprocessing import router as loan\_processing

from rolebased import router as rolebased\_router

from reporate import router as repo\_rate\_router

from announcement import router as announcement\_router, update\_announcements

import logging

import os

import warnings

warnings.filterwarnings("ignore", category=UserWarning)

warnings.filterwarnings("ignore", category=FutureWarning)

warnings.filterwarnings("ignore", message=".\*loading a serialized model.\*")

# configure logging to file + console

base\_dir = os.path.dirname(\_\_file\_\_)

log\_path = os.path.join(base\_dir, "system.log")

# root logger — this will catch uvicorn.\* logs too

logging.basicConfig(

    level=logging.INFO,

    format="%(levelname)s:     %(message)s",

    handlers=[

        logging.FileHandler(log\_path, mode="a", encoding="utf-8"),

        logging.StreamHandler(),  # still log to console

    ],

)

logger = logging.getLogger("rolebased")

logging.basicConfig(level=logging.INFO)

# ensure uvicorn loggers propagate into root

for name in ("uvicorn", "uvicorn.error", "uvicorn.access"):

    uvlogger = logging.getLogger(name)

    uvlogger.propagate = True

app = FastAPI(title="Modular Market API", version="2.0.0")

# custom validation error handler

@app.exception\_handler(RequestValidationError)

async def validation\_exception\_handler(request: Request, exc: RequestValidationError):

    return JSONResponse(status\_code=422, content={"detail": exc.errors(), "body": exc.body})

# CORS

app.add\_middleware(

    CORSMiddleware,

    allow\_origins=["\*"], allow\_credentials=True,

    allow\_methods=["\*"], allow\_headers=["\*"],

)

# static uploads

UPLOAD\_DIR = os.path.join(os.path.dirname(\_\_file\_\_), "uploads")

os.makedirs(UPLOAD\_DIR, exist\_ok=True)

app.mount("/uploads", StaticFiles(directory=UPLOAD\_DIR), name="uploads")

#routers

app.include\_router(negotiation\_router,      prefix="/api/negotiation", tags=["Negotiation"])

app.include\_router(investment\_router,       prefix="/api/investment",  tags=["Investment"])

app.include\_router(loan\_router,             prefix="/api/loan",        tags=["Loan Simulator"])

app.include\_router(markettrends\_router,     prefix="/api",             tags=["Market Trends"])

app.include\_router(expense\_router,                                     tags=["Expenses"])

app.include\_router(notification\_router,                                tags=["Notifications"])

app.include\_router(rewards\_router,                                     tags=["Rewards"])

app.include\_router(feedback\_router,         prefix="/feedback",        tags=["Feedback"])

app.include\_router(log\_router,                                         tags=["System Logs"])

app.include\_router(chatbot\_router,                                     tags=["Chatbot"])

app.include\_router(rolebased\_router)

app.include\_router(repo\_rate\_router)

app.include\_router(announcement\_router)

app.include\_router(auth\_router,                                        tags=["Authentication"])

app.include\_router(user\_router,             prefix="/api/users",       tags=["User Settings"])

app.include\_router(settings\_admin\_router,   prefix="/api/settings",    tags=["Settings Admin"])

app.include\_router(adminlogs\_router,                                   tags=["Admin Logs"])

app.include\_router(biometric\_router)

app.include\_router(predict\_router,          prefix="/api",            tags=["Loan Prediction"])

app.include\_router(emergency\_router)

app.include\_router(loan\_processing)

# Root endpoint

@app.get("/")

async def root():

    """

    Root endpoint with platform information

    """

    return {

        "platform": "🐺 ApexWolf Financial Services Platform",

        "version": "2.0.0",

        "description": "Advanced AI-powered financial assessment platform with role-based services",

        "status": "🟢 Online",

        "features": [

            "Role-based loan assessments",

            "AI-powered risk analysis",

            "Feature importance analysis",

            "Professional PDF reports",

            "Real-time predictions"

        ],

        "api\_docs": "/docs",

        "role\_based\_api": "/api/v1/role-based/",

        "supported\_roles": ["structured", "business", "selfeshtablished", "emerging"],

        "timestamp": datetime.utcnow().isoformat() + "Z"

    }

# Health check endpoint

@app.get("/health")

async def health\_check():

    """

    Platform health check

    """

    return {

        "status": "healthy",

        "platform": "ApexWolf Financial Services",

        "timestamp": datetime.utcnow().isoformat() + "Z",

        "components": {

            "api": " Operational",

            "role\_based\_services": " Operational",

            "ml\_models": " Loaded",

            "pdf\_generation": " Available"

        }

    }

# Error handlers

@app.exception\_handler(404)

async def not\_found\_handler(request, exc):

    return JSONResponse(

        status\_code=404,

        content={

            "error": "Endpoint not found",

            "message": "The requested endpoint does not exist",

            "available\_endpoints": {

                "platform\_info": "/",

                "health\_check": "/health",

                "api\_documentation": "/docs",

                "role\_based\_services": "/api/v1/role-based/"

            }

        }

    )

@app.exception\_handler(500)

async def internal\_error\_handler(request, exc):

    logger.error(f"Internal server error: {exc}")

    return JSONResponse(

        status\_code=500,

        content={

            "error": "Internal server error",

            "message": "An unexpected error occurred",

            "contact": "Please contact ApexWolf support"

        }

    )

# Shutdown event

@app.on\_event("shutdown")

async def shutdown\_event():

    """

    Application shutdown event

    """

    logger.info("🐺 ApexWolf Financial Services Platform shutting down...")

    logger.info(" Cleanup completed")

**AUTHENTICATION**

from fastapi import APIRouter, HTTPException

from fastapi.responses import JSONResponse

from pydantic import BaseModel, EmailStr

from db import get\_connection

router = APIRouter(prefix="/api")

class SignupRequest(BaseModel):

    full\_name: str

    email: EmailStr

    password: str

class LoginRequest(BaseModel):

    email: EmailStr

    password: str

class AdminLoginRequest(BaseModel):

    username: str

    password: str

@router.post("/auth/signup")

def signup(req: SignupRequest):

    conn = get\_connection()

    cursor = conn.cursor(dictionary=True)

    cursor.execute("SELECT id FROM app\_users WHERE email=%s", (req.email,))

    if cursor.fetchone():

        cursor.close()

        conn.close()

        raise HTTPException(400, "Email already registered")

    cursor.execute(

        "INSERT INTO app\_users (full\_name, email, password) VALUES (%s, %s, %s)",

        (req.full\_name, req.email, req.password)

    )

    conn.commit()

    user\_id = cursor.lastrowid

    cursor.execute("SELECT \* FROM app\_users WHERE id=%s", (user\_id,))

    user = cursor.fetchone()

    cursor.close()

    conn.close()

    return { "user": user }

@router.post("/auth/login")

def login(req: LoginRequest):

    conn = get\_connection()

    cursor = conn.cursor(dictionary=True)

    cursor.execute(

        "SELECT \* FROM app\_users WHERE email=%s AND password=%s",

        (req.email, req.password)

    )

    user = cursor.fetchone()

    if not user:

        cursor.close()

        conn.close()

        raise HTTPException(401, "Invalid credentials")

    user\_id = user['id']  # FIX: define user\_id now, before further queries

    # Fetch related user data

    cursor.execute("SELECT \* FROM expenses")

    expenses = cursor.fetchall()

    cursor.execute("SELECT \* FROM feedback WHERE user\_id=%s", (user\_id,))

    feedbacks = cursor.fetchall()

    cursor.execute("SELECT \* FROM notifications WHERE user\_id=%s", (user\_id,))

    notifications = cursor.fetchall()

    cursor.execute("SELECT \* FROM negotiation\_requests WHERE user\_id=%s", (user\_id,))

    negotiations = cursor.fetchall()

    cursor.execute("SELECT reward\_id FROM user\_redeemed\_rewards WHERE user\_id=%s", (user\_id,))

    redeemed = [r['reward\_id'] for r in cursor.fetchall()]

    cursor.execute("SELECT task\_id FROM user\_completed\_tasks WHERE user\_id=%s", (user\_id,))

    completed\_tasks = [t['task\_id'] for t in cursor.fetchall()]

    cursor.execute("SELECT \* FROM tasks")

    tasks = cursor.fetchall()

    cursor.execute("SELECT \* FROM rewards")

    rewards = cursor.fetchall()

    cursor.close()

    conn.close()

    response = JSONResponse({

        "msg": "Login successful",

        "user\_id": user["id"],

        "name": user["full\_name"],

        "email": user["email"]

    })

    response.set\_cookie(key="email", value=user["email"], httponly=True, samesite="lax")

    return {

        "user": user,

        "expenses": expenses,

        "feedbacks": feedbacks,

        "notifications": notifications,

        "negotiations": negotiations,

        "redeemed": redeemed,

        "completed\_tasks": completed\_tasks,

        "tasks": tasks,

        "rewards": rewards

    }

@router.post("/admin/login")

def admin\_login(req: AdminLoginRequest):

    conn = get\_connection()

    cursor = conn.cursor(dictionary=True)

    cursor.execute(

        "SELECT \* FROM admins WHERE username=%s AND password=%s",

        (req.username, req.password)

    )

    admin = cursor.fetchone()

    cursor.close()

    conn.close()

    if not admin:

        raise HTTPException(401, "Invalid credentials")

    return {"admin": admin}

**DATABASE**

import mysql.connector

def get\_connection():

    return mysql.connector.connect(

        host="127.0.0.1",

        port=3306,

        user="appuser",

        password="kavya",

        database="apexwolf"

    )

try:

    conn = get\_connection()

    print("Connected to apexwolf database successfully!")

    conn.close()

except Exception as e:

    print("Failed to connect to apexwolf database:", e)

**ELIGIBILITY PREDICTION**

from fastapi import APIRouter, HTTPException

from fastapi.responses import JSONResponse

from pydantic import BaseModel, field\_validator

from typing import Dict

import re

from db import get\_connection

import shap

import numpy as np

def parse\_number(s):

    if s is None:

        return 0.0

    if isinstance(s, (int, float)):

        return float(s)

    text = str(s).replace(",", "").strip().lower()

    m = re.match(r"^([\d\.]+)\s\*(k|thousand|lakh|crore)?$", text)

    if not m:

        try:

            return float(text)

        except ValueError:

            return 0.0

    num, unit = m.groups()

    val = float(num)

    if unit in ("k", "thousand"):

        return val \* 1\_000

    if unit == "lakh":

        return val \* 100\_000

    if unit == "crore":

        return val \* 10\_000\_000

    return val

class PredictionInput(BaseModel):

    user\_type: str

    inputs: Dict[str, str]

    @field\_validator("inputs")

    def validate\_inputs(cls, v):

        if not isinstance(v, dict):

            raise ValueError("Inputs must be a dict")

        return v

class WeightUpdate(BaseModel):

    feature: str

    option: str

    value: float

class PolynomialRegressionModel:

    def \_\_init\_\_(self):

        self.input\_features\_map = self.load\_input\_features()

        self.weights = self.load\_weights()

        self.formulas = self.load\_formulas()

        self.coefficients = self.parse\_coefficients()

    def load\_input\_features(self):

        return {

            "Full-Time Employment": [

                "Age", "Monthly Salary", "Years of Experience", "Job Tenure", "Credit Score",

                "Tax Status", "PF Balance", "Dependents", "Sector Type",

                "Education Level", "Insurance", "Job Stability Score"

            ],

            "Part-Time Employment": [

                "Age", "Monthly Income", "Weekly Hours", "Job Duration", "Secondary Income",

                "Credit Score", "Tax Filing Status", "Employer Type", "Skills",

                "Health Insurance", "Savings Level"

            ],

            "Apprenticeship": [

                "Age", "Stipend", "Program Duration", "Institution", "Grade", "Sponsored (Y/N)",

                "Living Costs", "Credit Score", "Parent Income", "Loan Purpose", "Digital Access",

                "Emergency Fund"

            ],

            "Traineeship": [

                "Age", "Program Length", "Affiliation", "Training Type", "Mentor Assigned",

                "Living Costs", "Transport Mode", "GPA", "Parent Support",

                "Certificate Status"

            ],

            "Internship": [

                "Age", "Paid/Unpaid", "Duration", "Stipend", "Parent Income", "GPA",

                "Credit Score", "Savings", "Loan Purpose", "Institution Ranking", "Career Interest",

                "Device Access"

            ],

            "Contract Employment": [

                "Age", "Income", "Contract Duration", "Employer Sector", "Prior Contracts",

                "Gaps", "Fixed Expenses", "Credit Score", "Emergency Fund",

                "Tax Records", "Project Count"

            ],

            "Seasonal Employment": [

                "Age", "Last Season Income", "Season Type", "Months Worked", "Off-Season Income",

                "Loan Purpose", "Credit Score", "Govt Scheme Eligibility", "Savings",

                "Family Dependents"

            ],

            "Casual Employment": [

                "Age", "Daily Wage", "Work Days/Month", "Role Type", "Income Consistency",

                "Credit Score", "Living Costs", "Health Status", "ID Proof",

                "Employer Type"

            ],

            "Employment on Commission": [

                "Age", "Monthly Earnings", "Product Sold", "Base Pay", "Sales Success Rate",

                "Credit Score", "Commission Volatility", "CRM Use", "Expenses",

                "Monthly Sales Target"

            ],

            "Probationary Employment": [

                "Age", "Salary", "Probation Period", "Employer Type", "Job Role",

                "Credit Score", "Insurance", "Expenses", "Future Job Confirmation",

                "Fixed Deductions"

            ],

            "Leased Employment": [

                "Age", "Salary", "Leasing Company", "Client Industry", "Contract Period",

                "Credit Score", "Hours Billed", "Expense Claims", "Job Changes",

                "Benefits Access"

            ],

            "Contingent Employment": [

                "Age", "Role (Freelancer/Consultant)", "Income", "Project Frequency",

                "Credit Score", "Platform Used", "Tools Owned", "Experience", "Work Category",

                "Tax Compliance"

            ],

            "Self-Employed": [

                "Age", "Business Type", "Monthly Revenue", "Tax Filing", "Credit Score",

                "Employees Count", "Operation Years", "Asset Owned",

                "Registration Status", "Sector"

            ],

            "Freelancer / Gig Worker": [

                "Age", "Skill Category", "Monthly Earnings", "Platform Used", "Clients Count",

                "Credit Score", "Ratings", "Internet Access", "Expenses", "Emergency Funds",

                "Tax Compliance"

            ],

            "Consultant / Independent Contractor": [

                "Age", "Field", "Hourly Rate", "Billing Hours", "Client Count",

                "Project Type", "Certifications", "Credit Score", "Contract Duration",

                "References"

            ],

            "Business Owner / Entrepreneur": [

                "Age", "Business Category", "Monthly Revenue", "Net Profit Margin", "Credit Score",

                "Employees", "Tax Record", "GST Score", "Assets",

                "Digital Adoption"

            ],

            "Retired (Pensioner)": [

                "Age", "Pension Income", "Savings", "Credit Score",

                "Health Costs", "Insurance", "Dependents", "Property Ownership", "Spouse Income",

                "Past Employment"

            ],

            "Homemaker": [

                "Age", "Spouse Job Type", "Household Income", "Loan Co-signer", "Credit Score",

                "Family Expenses", "Dependents", "Property Ownership", "Insurance", "ID Proof"

            ],

            "Unemployed – Actively Looking": [

                "Age", "Last Job Duration", "Gap Length", "Applications/Week", "Credit Score",

                "Emergency Fund", "Expected Salary", "Skills Training (Y/N)"

            ],

            "Unemployed – Not Looking": [

                "Age", "Career Gap Reason", "Past Job", "Credit Score", "Supporter Income",

                "Expenses", "Training Plan", "Loan Co-signer", "Emergency Fund", "Property Ownership"

            ],

            "Student": [

                "Age", "Education Level", "GPA", "Institution", "Scholarship",

                "Family Income", "Part-Time Job", "Living Expenses", "Device Access", "Co-signer",

                "Credit Score"

            ],

            "Receiving Government Assistance": [

                "Age", "Assistance Type", "Monthly Income", "Credit Score",

                "Housing", "Dependents", "Tax Status", "Savings", "Government ID",

                "Loan History"

            ],

            "Volunteer Work (Unpaid)": [

                "Age", "Volunteer Hours", "Org Type", "Duration", "Skills",

                "Living Costs", "Credit Score", "Sponsor (Y/N)", "Family Support", "Loan Purpose",

                "Expenses"

            ],

            "Investor": [

                "Age", "Income Type", "Monthly Income", "Portfolio Value", "Credit Score",

                "Property Owned", "Asset Types", "Risk Score", "Tax Filing", "Loan Purpose",

                "Volatility Buffer"

            ]

        }

    def load\_weights(self) -> Dict[str, Dict[str, float]]:

        # Load feature weights (default: all 1.0)

        return { // Default Weights will be present here }

    def load\_formulas(self):

        formulas = {}

        for user\_type, features in self.input\_features\_map.items():

            formula = "eligible\_amount = " + " + ".join(

                [f"w{i+1}\*{feature.lower().replace(' ', '\_').replace('(', '').replace(')', '')}" for i, feature in enumerate(features)]

            )

            formulas[user\_type] = formula

        return formulas

    def parse\_coefficients(self):

        coefficients = {}

        for user\_type, features in self.input\_features\_map.items():

            coef\_dict = {}

            for feat in features:

            # If the feature has weights, use its default highest weighted option

                if feat in self.weights:

                    options = self.weights[feat]

                # Choose the average or max weight as default coefficient

                    coef\_value = sum(options.values()) / len(options)

                    coef\_dict[feat] = coef\_value

                else:

                    coef\_dict[feat] = 1.0  # fallback

            coefficients[user\_type] = coef\_dict

        return coefficients

    def normalize\_inputs(self, user\_type: str, inputs: Dict[str, str]) -> Dict[str, float]:

        norm: Dict[str, float] = {}

        for feat, raw in inputs.items():

            if feat in self.weights and raw in self.weights[feat]:

                norm[feat] = self.weights[feat][raw]

            else:

                norm[feat] = parse\_number(raw)

        return norm

    def predict(self, user\_type: str, inputs: Dict[str, str]) -> Dict[str, float]:

        import math

        norm = self.normalize\_inputs(user\_type, inputs)

        coefs = {

            f: self.weights.get(f, {}).get(inputs.get(f, ""), 1.0)

            for f in norm

        }

        eligible\_amount = sum(norm[f] \* coefs.get(f, 1.0) for f in norm)

        credit\_score = norm.get("Credit Score", 700)

        if credit\_score >= 900:

            base\_rate = 7.0

        elif credit\_score >= 800:

            base\_rate = 8.5

        elif credit\_score >= 700:

            base\_rate = 10.0

        else:

            base\_rate = 12.5

        discount = min(eligible\_amount \* 0.00001, 5.0)

        effective\_rate = base\_rate - discount

        default\_tenure = int(norm.get("Loan Tenure", 12))

        tenure = default\_tenure

        monthly\_rate = effective\_rate / 1200

        monthly\_salary = norm.get("Monthly Salary", 0)

        max\_emi = 0.25 \* monthly\_salary if monthly\_salary else float('inf')

        if tenure == 0 or monthly\_rate == 0:

            emi = eligible\_amount / max(tenure, 1)

        else:

            emi = (

                eligible\_amount

                \* monthly\_rate

                \* (1 + monthly\_rate) \*\* tenure

            ) / (

                (1 + monthly\_rate) \*\* tenure - 1

            )

            if emi > max\_emi:

                try:

                    A = max\_emi

                    P = eligible\_amount

                    r = monthly\_rate

                    n = math.log(A / (A - P \* r)) / math.log(1 + r)

                    tenure = int(math.ceil(n))

                    emi = A

                except:

                    tenure = default\_tenure

                    emi = eligible\_amount / max(tenure, 1)

        risk = (

            "Low" if eligible\_amount > len(norm) \* 70\_000

            else "Moderate" if eligible\_amount > len(norm) \* 40\_000

            else "High"

        )

        return {

            "eligible\_amount": round(eligible\_amount, 2),

            "base\_rate": round(base\_rate, 2),

            "discount": round(discount, 2),

            "effective\_rate": round(effective\_rate, 2),

            "tenure": tenure,

            "emi": round(emi, 2),

            "risk": risk

        }

model = PolynomialRegressionModel()

predict\_router = APIRouter()

@predict\_router.post("/predict")

def predict(input\_data: PredictionInput):

    try:

        result = model.predict(input\_data.user\_type, input\_data.inputs)

        norm = model.normalize\_inputs(input\_data.user\_type, input\_data.inputs)

        feature\_names = list(norm.keys())

        X = np.array([[norm[f] for f in feature\_names]])

        def f(x):

            return np.sum(x \* np.array([model.coefficients[input\_data.user\_type].get(f, 1.0) for f in feature\_names]), axis=1)

        explainer = shap.KernelExplainer(f, np.zeros((1, X.shape[1])))

        shap\_vals = explainer.shap\_values(X)[0]

        shap\_dict = {

            feature\_names[i]: round(float(shap\_vals[i]), 2)

            for i in range(len(feature\_names))

        }

        result["shap\_values"] = shap\_dict

        return JSONResponse(content=result)

    except Exception as e:

        raise HTTPException(status\_code=400, detail=str(e))

from fastapi.responses import JSONResponse

from fastapi import HTTPException

from pydantic import BaseModel, field\_validator

class WeightUpdate(BaseModel):

    option: str

    value: float

    @field\_validator('value', mode='before')

    def convert\_str\_to\_float(cls, v):

        try:

            return float(v)

        except Exception:

            raise ValueError('Input should be a valid number, unable to parse string as a number')

@predict\_router.get("/loan/weights")

def get\_weight\_features():

    return list(model.weights.keys())

@predict\_router.get("/loan/weights/{feature:path}")

def get\_weight\_for\_feature(feature: str):

    if feature in model.weights:

        return model.weights[feature]

    raise HTTPException(status\_code=404, detail="Feature not found.")

@predict\_router.post("/loan/weights/{feature:path}")

def update\_weight\_for\_feature(feature: str, data: WeightUpdate):  # <-- CHANGE IS HERE

    option = data.option

    value = data.value

    if feature in model.weights and option in model.weights[feature]:

        model.weights[feature][option] = value

        return JSONResponse(content={"status": "ok"})

    raise HTTPException(status\_code=404, detail="Feature/Option not found.")

**ROLE BASED PREDICTION**

from fastapi import APIRouter, HTTPException, BackgroundTasks

from pydantic import BaseModel, Field, validator

from typing import Dict, Any, Optional, List

import joblib

import numpy as np

import pdfkit

import io

import os

from datetime import datetime, timezone

from fastapi.responses import StreamingResponse

import logging

import json

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

# Create router

router = APIRouter(prefix="/api/v1/role-based", tags=["Role-Based Financial Services"])

# Constants

USER\_TYPES = ["structured", "business", "selfestablished", "emerging"]

MODEL\_TYPES = ["interest\_model", "risk\_model", "loan\_tenure\_model", "tenure\_bucket\_model"]

USER\_TYPE\_FEATURES = {

    "structured": [

        "Age", "Income\_Level", "Loan\_Amount", "Credit\_Score", "Existing\_FD\_Amount",

        "FD\_Tenure\_Months", "Years\_of\_Experience", "Job\_Sector\_Government",

        "Job\_Sector\_PSU", "Job\_Sector\_Private", "Job\_Sector\_Startup"

    ],

    "business": [

        "Years\_in\_Operation", "Annual\_Revenue", "Loan\_Amount", "Monthly\_Revenue\_Variance",

        "Credit\_Score", "GST\_Return\_Score", "ITR\_Compliance\_Score", "Banking\_History\_Score",

        "Collateral\_Value", "Collateral\_Coverage\_Ratio", "Revenue\_to\_Loan\_Ratio",

        "Num\_of\_Defaults", "Market\_Trend\_Impact"

    ],

    "selfestablished": [

        "Age", "Net\_Worth\_Cr", "Annual\_Income", "Business\_Interest",

        "Investment\_Portfolio\_Diversity", "Real\_Estate\_Value", "Existing\_FD\_Amount",

        "Loan\_Amount", "Credit\_Score", "Luxury\_Expenditure\_%", "Bank\_Relationship",

        "Repayment\_History", "Market\_Trend\_Impact", "Debt\_to\_Asset\_Ratio",

        "Spending\_to\_Income\_Ratio"

    ],

    "emerging": [

        "Age", "Monthly\_Income", "Existing\_Loan", "Existing\_FD\_Amount",

        "Loan\_Amount", "Credit\_Score", "Financial\_Literacy\_Score",

        "Has\_Dependent\_Children", "Marital\_Status", "Employment\_Status",

        "Bank\_Affiliation", "Repayment\_Behavior", "Market\_Trend\_Impact"

    ]

}

# Pydantic Models

class PredictionRequest(BaseModel):

    user\_type: str = Field(..., description="Type of user: structured, business, selfeshtablished, emerging")

    user\_data: Dict[str, Any] = Field(..., description="User financial data")

    @validator('user\_type')

    def validate\_user\_type(cls, v):

        if v not in USER\_TYPES:

            raise ValueError(f"user\_type must be one of: {USER\_TYPES}")

        return v

class PredictionResponse(BaseModel):

    base\_interest\_rate: Optional[float] = Field(None, description="Base interest rate (%)")

    discounted\_interest\_rate: Optional[float] = Field(None, description="Discounted interest rate (%)")

    loan\_tenure: Optional[int] = Field(None, description="Loan tenure in months")

    risk\_rating: Optional[str] = Field(None, description="Risk rating: Low, Medium, High")

    confidence\_score: Optional[float] = Field(None, description="Model confidence score")

    timestamp: datetime = Field(default\_factory=lambda: datetime.now(timezone.utc))

class ExplanationResponse(BaseModel):

    feature\_importance: Dict[str, float] = Field(..., description="Feature importance scores")

    recommendations: List[str] = Field(default\_factory=list, description="Personalized recommendations")

    timestamp: datetime = Field(default\_factory=lambda: datetime.now(timezone.utc))

class ReportRequest(BaseModel):

    user\_profile: Dict[str, Any] = Field(..., description="User profile information")

    prediction: Dict[str, Any] = Field(..., description="Prediction results")

    explanation: Dict[str, Any] = Field(..., description="Model explanation")

    include\_charts: bool = Field(default=True, description="Include visual charts in report")

# Global variables

user\_models = {}

model\_metadata = {}

def load\_models():

    """Load all ML models and their metadata"""

    global user\_models, model\_metadata

    models\_dir = os.path.join(os.path.dirname(\_\_file\_\_), "models")

    for user\_type in USER\_TYPES:

        user\_models[user\_type] = {}

        model\_metadata[user\_type] = {}

        for model\_type in MODEL\_TYPES:

            model\_file = f"{user\_type}\_{model\_type}.pkl"

            model\_path = os.path.join(models\_dir, model\_file)

            try:

                if os.path.exists(model\_path):

                    user\_models[user\_type][model\_type] = joblib.load(model\_path)

                    # Try to load metadata if available

                    metadata\_path = os.path.join(models\_dir, f"{user\_type}\_{model\_type}\_metadata.json")

                    try:

                        with open(metadata\_path, 'r') as f:

                            model\_metadata[user\_type][model\_type] = json.load(f)

                    except FileNotFoundError:

                        model\_metadata[user\_type][model\_type] = {

                            "accuracy": "N/A",

                            "last\_trained": "N/A",

                            "model\_file": model\_file

                        }

                    logger.info(f" Loaded model: {model\_file}")

                else:

                    logger.warning(f" Model file not found: {model\_path}")

                    user\_models[user\_type][model\_type] = None

                    model\_metadata[user\_type][model\_type] = {"error": f"File not found: {model\_file}"}

            except Exception as e:

                user\_models[user\_type][model\_type] = None

                model\_metadata[user\_type][model\_type] = {"error": str(e)}

                logger.error(f" Failed to load model {model\_file}: {e}")

# Initialize models on startup

load\_models()

# In-memory storage (replace with database in production)

user\_sessions: Dict[str, Dict[str, Any]] = {}

def prepare\_features(user\_type: str, data: Dict[str, Any]) -> Dict[str, Any]:

    """Prepare and validate features for model input"""

    expected\_features = USER\_TYPE\_FEATURES.get(user\_type, [])

    features = {}

    for feature in expected\_features:

        value = data.get(feature, 0)

        # Handle boolean values

        if isinstance(value, bool):

            features[feature] = 1 if value else 0

        # Handle categorical variables

        elif feature == "Marital\_Status":

            mapping = {"single": 0, "married": 1, "divorced": 2, "widowed": 3}

            features[feature] = mapping.get(str(value).lower(), 0)

        elif feature == "Employment\_Status":

            mapping = {"structured": 0, "self-employed": 1, "business": 2, "unemployed": 3, "retired": 4}

            features[feature] = mapping.get(str(value).lower(), 0)

        # Handle numeric values

        else:

            try:

                features[feature] = float(value)

            except (ValueError, TypeError):

                features[feature] = 0

                logger.warning(f"⚠️ Invalid value for feature {feature}: {value}, using default 0")

    return features

def create\_model\_input(user\_type: str, features: Dict[str, Any]) -> np.ndarray:

    """Create numpy array input for model prediction"""

    expected\_features = USER\_TYPE\_FEATURES.get(user\_type, [])

    input\_array = [features.get(feature, 0) for feature in expected\_features]

    return np.array([input\_array])

def normalize\_interest\_rate(rate: Optional[float]) -> Optional[float]:

    """Normalize interest rate to realistic bounds"""

    if rate is None:

        return None

    rate = round(rate, 2)

    # Apply normalization logic for unrealistic rates

    if rate > 15:

        if rate <= 20:

            return round(7 + (rate - 15) \* (2/5), 2)

        elif rate <= 25:

            return round(9 + (rate - 20) \* (3/5), 2)

        return round(12 + min(3, (rate - 25) \* 0.1), 2)

    # Ensure minimum rate

    return max(3.0, rate)

def normalize\_loan\_tenure(tenure: Optional[float]) -> Optional[int]:

    """Normalize loan tenure to realistic bounds"""

    if tenure is None:

        return None

    tenure = round(tenure)

    # Apply bounds

    if tenure < 3:

        return 3

    if tenure > 240:

        if tenure > 500:

            return min(240, int(100 + 20 \* np.log10(tenure/100)))

        return 240

    return tenure

def calculate\_confidence\_score(user\_type: str, features: Dict[str, Any]) -> float:

    """Calculate model confidence based on feature completeness and quality"""

    expected\_features = USER\_TYPE\_FEATURES.get(user\_type, [])

    # Feature completeness score

    provided\_features = sum(1 for f in expected\_features if features.get(f, 0) != 0)

    completeness\_score = provided\_features / len(expected\_features)

    # Feature quality score (simplified)

    quality\_score = 0.8  # Base quality score

    # Adjust based on critical features

    critical\_features = ["Credit\_Score", "Loan\_Amount"]

    for feature in critical\_features:

        if feature in features and features[feature] > 0:

            quality\_score += 0.1

    return min(1.0, (completeness\_score \* 0.7 + quality\_score \* 0.3))

def generate\_recommendations(user\_type: str, features: Dict[str, Any], prediction: Dict[str, Any]) -> List[str]:

    """Generate personalized recommendations based on user profile and predictions"""

    recommendations = []

    # Credit score recommendations

    credit\_score = features.get("Credit\_Score", 0)

    if credit\_score < 650:

        recommendations.append("💡 Consider improving your credit score to get better interest rates")

    elif credit\_score > 750:

        recommendations.append("⭐ Your excellent credit score qualifies you for premium rates")

    # Risk-based recommendations

    risk\_rating = prediction.get("risk\_rating", "Medium")

    if risk\_rating == "High":

        recommendations.append("🔒 Consider providing additional collateral to reduce risk assessment")

    elif risk\_rating == "Low":

        recommendations.append(" You qualify for our best rates and flexible terms")

    # User type specific recommendations

    if user\_type == "business":

        if features.get("GST\_Return\_Score", 0) < 70:

            recommendations.append("📊 Improve GST compliance to enhance loan eligibility")

        if features.get("Banking\_History\_Score", 0) > 80:

            recommendations.append("🏦 Your strong banking relationship offers additional benefits")

    elif user\_type == "structured":

        if features.get("Years\_of\_Experience", 0) < 2:

            recommendations.append("👥 Consider a co-applicant to strengthen your application")

        if features.get("Job\_Sector\_Government", 0) == 1:

            recommendations.append("🏛️ Government employees enjoy special interest rate benefits")

    elif user\_type == "selfeshtablished":

        if features.get("Investment\_Portfolio\_Diversity", 0) > 70:

            recommendations.append("💼 Your diversified portfolio qualifies for premium banking services")

    elif user\_type == "emerging":

        if features.get("Financial\_Literacy\_Score", 0) > 75:

            recommendations.append("📚 Your financial literacy qualifies for emerging entrepreneur schemes")

    return recommendations

def predict\_financial\_metrics(user\_type: str, data: Dict[str, Any]) -> Dict[str, Any]:

    """Enhanced prediction function with confidence scoring"""

    models = user\_models.get(user\_type)

    if not models:

        logger.warning(f"⚠️ No models available for user type: {user\_type}")

        return {

            "base\_interest\_rate": 7.5,

            "discounted\_interest\_rate": 6.5,

            "loan\_tenure": 60,

            "risk\_rating": "Medium",

            "confidence\_score": 0.5

        }

    features = prepare\_features(user\_type, data)

    input\_array = create\_model\_input(user\_type, features)

    confidence\_score = calculate\_confidence\_score(user\_type, features)

    results = {}

    model\_mapping = {

        "interest\_model": "base\_interest\_rate",

        "tenure\_bucket\_model": "discounted\_interest\_rate",

        "loan\_tenure\_model": "loan\_tenure",

        "risk\_model": "risk\_rating"

    }

    for model\_type, result\_key in model\_mapping.items():

        model = models.get(model\_type)

        if model is None:

            logger.warning(f"⚠️ Model {model\_type} not available for {user\_type}")

            results[result\_key] = None

            continue

        try:

            prediction = model.predict(input\_array)

            if model\_type == "risk\_model":

                risk\_mapping = {0: "Low", 1: "Medium", 2: "High"}

                results[result\_key] = risk\_mapping.get(int(round(prediction[0])), "Unknown")

            else:

                results[result\_key] = float(prediction[0])

        except Exception as e:

            logger.error(f" Prediction failed for {model\_type}: {e}")

            results[result\_key] = None

    return {

        "base\_interest\_rate": normalize\_interest\_rate(results["base\_interest\_rate"]),

        "discounted\_interest\_rate": normalize\_interest\_rate(results["discounted\_interest\_rate"]),

        "loan\_tenure": normalize\_loan\_tenure(results["loan\_tenure"]),

        "risk\_rating": results["risk\_rating"],

        "confidence\_score": round(confidence\_score, 2)

    }

def generate\_enhanced\_html\_report(user\_profile: Dict, prediction: Dict, explanation: Dict, include\_charts: bool = True, user\_role: str = "Client") -> str:

    """Generate enhanced HTML report with better styling and role-based customization"""

    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S UTC")

    html = f"""

    <!DOCTYPE html>

    <html lang="en">

    <head>

        <meta charset="UTF-8">

        <meta name="viewport" content="width=device-width, initial-scale=1.0">

        <title>ApexWolf Financial Assessment Report - {user\_role}</title>

        <style>

            body {{

                font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

                line-height: 1.6;

                margin: 0;

                padding: 20px;

                background-color: #f8f9fa;

                color: #333;

            }}

            .container {{

                max-width: 800px;

                margin: 0 auto;

                background: white;

                padding: 30px;

                border-radius: 10px;

                box-shadow: 0 0 20px rgba(0,0,0,0.1);

            }}

            .header {{

                text-align: center;

                margin-bottom: 30px;

                padding-bottom: 20px;

                border-bottom: 3px solid #007bff;

            }}

            .header h1 {{

                color: #007bff;

                margin: 0;

                font-size: 28px;

            }}

            .header p {{

                color: #666;

                margin: 5px 0 0 0;

            }}

            .section {{

                margin: 30px 0;

            }}

            .section h2 {{

                color: #007bff;

                border-bottom: 2px solid #e9ecef;

                padding-bottom: 10px;

                margin-bottom: 20px;

            }}

            .info-grid {{

                display: grid;

                grid-template-columns: 1fr 1fr;

                gap: 20px;

                margin-bottom: 20px;

            }}

            .info-card {{

                background: #f8f9fa;

                padding: 15px;

                border-radius: 8px;

                border-left: 4px solid #007bff;

            }}

            .info-card h3 {{

                margin: 0 0 10px 0;

                color: #007bff;

                font-size: 16px;

            }}

            .info-card p {{

                margin: 0;

                font-weight: bold;

                font-size: 18px;

            }}

            table {{

                width: 100%;

                border-collapse: collapse;

                margin: 20px 0;

                background: white;

            }}

            th, td {{

                padding: 12px;

                text-align: left;

                border-bottom: 1px solid #ddd;

            }}

            th {{

                background-color: #007bff;

                color: white;

                font-weight: 600;

            }}

            tr:nth-child(even) {{

                background-color: #f8f9fa;

            }}

            .risk-low {{ color: #28a745; font-weight: bold; }}

            .risk-medium {{ color: #ffc107; font-weight: bold; }}

            .risk-high {{ color: #dc3545; font-weight: bold; }}

            .recommendations {{

                background: #e8f4f8;

                padding: 20px;

                border-radius: 8px;

                border-left: 4px solid #17a2b8;

            }}

            .recommendations ul {{

                margin: 10px 0;

                padding-left: 20px;

            }}

            .recommendations li {{

                margin: 8px 0;

            }}

            .footer {{

                text-align: center;

                margin-top: 30px;

                padding-top: 20px;

                border-top: 1px solid #ddd;

                color: #666;

                font-size: 12px;

            }}

            .chart-placeholder {{

                background: #f8f9fa;

                height: 200px;

                border: 2px dashed #dee2e6;

                display: flex;

                align-items: center;

                justify-content: center;

                color: #6c757d;

                margin: 20px 0;

                border-radius: 8px;

            }}

        </style>

    </head>

    <body>

        <div class="container">

            <div class="header">

                <h1>🐺 ApexWolf Financial Assessment Report</h1>

                <p>Role-Based Financial Analysis & Recommendations for {user\_role}</p>

                <p><strong>Generated on:</strong> {timestamp}</p>

            </div>

            <div class="section">

                <h2>📊 Financial Prediction Summary</h2>

                <div class="info-grid">

                    <div class="info-card">

                        <h3>Base Interest Rate</h3>

                        <p>{prediction.get('base\_interest\_rate', 'N/A')}%</p>

                    </div>

                    <div class="info-card">

                        <h3>Discounted Rate</h3>

                        <p>{prediction.get('discounted\_interest\_rate', 'N/A')}%</p>

                    </div>

                    <div class="info-card">

                        <h3>Loan Tenure</h3>

                        <p>{prediction.get('loan\_tenure', 'N/A')} months</p>

                    </div>

                    <div class="info-card">

                        <h3>Risk Rating</h3>

                        <p class="risk-{prediction.get('risk\_rating', 'medium').lower()}">{prediction.get('risk\_rating', 'N/A')}</p>

                    </div>

                </div>

            </div>

            <div class="section">

                <h2>👤 User Profile</h2>

                <table>

                    <thead>

                        <tr>

                            <th>Attribute</th>

                            <th>Value</th>

                        </tr>

                    </thead>

                    <tbody>

    """

    for key, value in user\_profile.items():

        formatted\_key = key.replace('\_', ' ').title()

        html += f"<tr><td>{formatted\_key}</td><td>{value}</td></tr>"

    html += """

                    </tbody>

                </table>

            </div>

            <div class="section">

                <h2>🔍 Feature Importance Analysis</h2>

                <table>

                    <thead>

                        <tr>

                            <th>Feature</th>

                            <th>Impact Score</th>

                            <th>Influence</th>

                        </tr>

                    </thead>

                    <tbody>

    """

    for feature, score in explanation.items():

        impact = "Positive" if score > 0 else "Negative" if score < 0 else "Neutral"

        formatted\_feature = feature.replace('\_', ' ').title()

        html += f"<tr><td>{formatted\_feature}</td><td>{score:.3f}</td><td>{impact}</td></tr>"

    # Generate recommendations

    recommendations = [

        "💡 Consider improving credit score for better rates",

        "📈 Maintain stable income documentation",

        "💰 Review loan amount based on current financial capacity",

        "📊 Explore options for reducing existing debt burden"

    ]

    html += f"""

                    </tbody>

                </table>

            </div>

            <div class="section">

                <div class="recommendations">

                    <h2>💡 Personalized Recommendations</h2>

                    <ul>

    """

    for rec in recommendations:

        html += f"<li>{rec}</li>"

    if include\_charts:

        html += f"""

            <div class="section">

                <h2>📈 Visual Analysis - {user\_role} Specific Metrics</h2>

                <div class="chart-placeholder">

                    Risk Assessment Chart - {user\_role} feature importance visualization would appear here

                </div>

            </div>

        """

    html += f"""

                    </ul>

                </div>

            </div>

            <div class="footer">

                <p><strong>Disclaimer:</strong> This report is generated by ApexWolf AI models and should be used for reference only.</p>

                <p>For detailed financial advice, please consult with our specialized {user\_role.lower()} financial advisors.</p>

                <p><strong>Confidence Score:</strong> {prediction.get('confidence\_score', 'N/A')}</p>

                <p><strong>ApexWolf Role-Based Financial Services</strong> - Tailored solutions for every professional profile</p>

            </div>

        </div>

    </body>

    </html>

    """

    return html

# === ROLE-BASED API ENDPOINTS ===

# Structured Employee Endpoints

@router.post("/structured-employee/loan-assessment", response\_model=PredictionResponse)

async def assess\_structured\_employee\_loan(request: PredictionRequest):

    """

    Assess loan terms for structured employees including interest rates, tenure, and risk

    """

    try:

        if request.user\_type != "structured":

            raise HTTPException(status\_code=400, detail="This endpoint is for structured employees only")

        prediction\_result = predict\_financial\_metrics(request.user\_type, request.user\_data)

        return PredictionResponse(

            base\_interest\_rate=prediction\_result.get("base\_interest\_rate"),

            discounted\_interest\_rate=prediction\_result.get("discounted\_interest\_rate"),

            loan\_tenure=prediction\_result.get("loan\_tenure"),

            risk\_rating=prediction\_result.get("risk\_rating"),

            confidence\_score=prediction\_result.get("confidence\_score")

        )

    except Exception as e:

        logger.error(f" Structured employee assessment error: {e}")

        raise HTTPException(status\_code=500, detail=f"Assessment failed: {str(e)}")

@router.post("/structured-employee/feature-analysis", response\_model=ExplanationResponse)

async def analyze\_structured\_employee\_features(request: PredictionRequest):

    """

    Analyze feature importance for structured employee loan applications

    """

    try:

        if request.user\_type != "structured":

            raise HTTPException(status\_code=400, detail="This endpoint is for structured employees only")

        user\_type = request.user\_type

        features = prepare\_features(user\_type, request.user\_data)

        # Enhanced explanation logic

        explanation = {}

        for feature, value in features.items():

            if feature == "Credit\_Score":

                explanation[feature] = min(0.5, value / 900) if value > 0 else -0.2

            elif "Income" in feature or "Revenue" in feature:

                explanation[feature] = min(0.4, value / 100000) if value > 0 else -0.15

            elif feature == "Loan\_Amount":

                explanation[feature] = -min(0.3, value / 500000) if value > 0 else 0

            elif feature == "Age":

                if 30 <= value <= 50:

                    explanation[feature] = 0.15

                elif 25 <= value < 30 or 50 < value <= 60:

                    explanation[feature] = 0.05

                else:

                    explanation[feature] = -0.1

            elif "Experience" in feature or "Operation" in feature:

                explanation[feature] = min(0.2, value / 10) if value > 0 else -0.1

            else:

                explanation[feature] = 0.05 if value > 0 else -0.05

        # Generate predictions for recommendations

        prediction\_result = predict\_financial\_metrics(user\_type, request.user\_data)

        recommendations = generate\_recommendations(user\_type, features, prediction\_result)

        return ExplanationResponse(

            feature\_importance=explanation,

            recommendations=recommendations

        )

    except Exception as e:

        logger.error(f" Structured employee feature analysis error: {e}")

        raise HTTPException(status\_code=500, detail=f"Feature analysis failed: {str(e)}")

@router.post("/structured-employee/generate-report")

async def generate\_structured\_employee\_report(request: ReportRequest):

    """

    Generate comprehensive PDF financial report for structured employees

    """

    try:

        html\_content = generate\_enhanced\_html\_report(

            user\_profile=request.user\_profile,

            prediction=request.prediction,

            explanation=request.explanation,

            include\_charts=request.include\_charts,

            user\_role="Structured Employee"

        )

        # PDF configuration

        options = {

            'page-size': 'A4',

            'margin-top': '0.75in',

            'margin-right': '0.75in',

            'margin-bottom': '0.75in',

            'margin-left': '0.75in',

            'encoding': "UTF-8",

            'no-outline': None,

            'enable-local-file-access': None

        }

        try:

            config = pdfkit.configuration(wkhtmltopdf="/usr/bin/wkhtmltopdf")

            pdf\_content = pdfkit.from\_string(html\_content, False, options=options, configuration=config)

        except:

            # Fallback for systems without wkhtmltopdf

            logger.warning("⚠️ wkhtmltopdf not found, using alternative PDF generation")

            pdf\_content = html\_content.encode('utf-8')  # Fallback to HTML

        filename = f"apexwolf\_structured\_employee\_report\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.pdf"

        return StreamingResponse(

            io.BytesIO(pdf\_content),

            media\_type="application/pdf",

            headers={"Content-Disposition": f"attachment; filename={filename}"}

        )

    except Exception as e:

        logger.error(f" Structured employee report generation error: {e}")

        raise HTTPException(status\_code=500, detail=f"Report generation failed: {str(e)}")

# Business Owner Endpoints

@router.post("/business-owner/loan-assessment", response\_model=PredictionResponse)

async def assess\_business\_owner\_loan(request: PredictionRequest):

    """

    Assess loan terms for business owners with revenue and compliance analysis

    """

    try:

        if request.user\_type != "business":

            raise HTTPException(status\_code=400, detail="This endpoint is for business owners only")

        prediction\_result = predict\_financial\_metrics(request.user\_type, request.user\_data)

        return PredictionResponse(

            base\_interest\_rate=prediction\_result.get("base\_interest\_rate"),

            discounted\_interest\_rate=prediction\_result.get("discounted\_interest\_rate"),

            loan\_tenure=prediction\_result.get("loan\_tenure"),

            risk\_rating=prediction\_result.get("risk\_rating"),

            confidence\_score=prediction\_result.get("confidence\_score")

        )

    except Exception as e:

        logger.error(f" Business loan assessment error: {e}")

        raise HTTPException(status\_code=500, detail=f"Assessment failed: {str(e)}")

@router.post("/business-owner/feature-analysis", response\_model=ExplanationResponse)

async def analyze\_business\_owner\_features(request: PredictionRequest):

    """

    Analyze feature importance for business owner loan applications

    """

    try:

        if request.user\_type != "business":

            raise HTTPException(status\_code=400, detail="This endpoint is for business owners only")

        user\_type = request.user\_type

        features = prepare\_features(user\_type, request.user\_data)

        # Business-specific explanation logic

        explanation = {}

        for feature, value in features.items():

            if feature == "Credit\_Score":

                explanation[feature] = min(0.5, value / 900) if value > 0 else -0.2

            elif "Revenue" in feature:

                explanation[feature] = min(0.4, value / 1000000) if value > 0 else -0.15

            elif feature == "GST\_Return\_Score":

                explanation[feature] = min(0.3, value / 100) if value > 0 else -0.2

            elif feature == "ITR\_Compliance\_Score":

                explanation[feature] = min(0.25, value / 100) if value > 0 else -0.15

            elif feature == "Years\_in\_Operation":

                explanation[feature] = min(0.2, value / 20) if value > 0 else -0.1

            elif feature == "Collateral\_Coverage\_Ratio":

                explanation[feature] = min(0.3, value / 2) if value > 1 else -0.1

            elif feature == "Num\_of\_Defaults":

                explanation[feature] = -min(0.4, value \* 0.1) if value > 0 else 0.1

            else:

                explanation[feature] = 0.05 if value > 0 else -0.05

        # Generate predictions for recommendations

        prediction\_result = predict\_financial\_metrics(user\_type, request.user\_data)

        recommendations = generate\_recommendations(user\_type, features, prediction\_result)

        return ExplanationResponse(

            feature\_importance=explanation,

            recommendations=recommendations

        )

    except Exception as e:

        logger.error(f" Business owner feature analysis error: {e}")

        raise HTTPException(status\_code=500, detail=f"Feature analysis failed: {str(e)}")

@router.post("/business-owner/generate-report")

async def generate\_business\_owner\_report(request: ReportRequest):

    """

    Generate comprehensive PDF financial report for business owners

    """

    try:

        html\_content = generate\_enhanced\_html\_report(

            user\_profile=request.user\_profile,

            prediction=request.prediction,

            explanation=request.explanation,

            include\_charts=request.include\_charts,

            user\_role="Business Owner"

        )

        # PDF configuration

        options = {

            'page-size': 'A4',

            'margin-top': '0.75in',

            'margin-right': '0.75in',

            'margin-bottom': '0.75in',

            'margin-left': '0.75in',

            'encoding': "UTF-8",

            'no-outline': None,

            'enable-local-file-access': None

        }

        try:

            config = pdfkit.configuration(wkhtmltopdf="/usr/bin/wkhtmltopdf")

            pdf\_content = pdfkit.from\_string(html\_content, False, options=options, configuration=config)

        except:

            logger.warning("⚠️ wkhtmltopdf not found, using alternative PDF generation")

            pdf\_content = html\_content.encode('utf-8')

        filename = f"apexwolf\_business\_owner\_report\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.pdf"

        return StreamingResponse(

            io.BytesIO(pdf\_content),

            media\_type="application/pdf",

            headers={"Content-Disposition": f"attachment; filename={filename}"}

        )

    except Exception as e:

        logger.error(f" Business owner report generation error: {e}")

        raise HTTPException(status\_code=500, detail=f"Report generation failed: {str(e)}")

# SELFESHTABLISHED Client Endpoints

@router.post("/selfeshtablished-client/loan-assessment", response\_model=PredictionResponse)

async def assess\_selfeshtablished\_client\_loan(request: PredictionRequest):

    """

    Assess loan terms for High Net Worth Individuals with portfolio analysis

    """

    try:

        if request.user\_type != "selfeshtablished":

            raise HTTPException(status\_code=400, detail="This endpoint is for SELFESHTABLISHED clients only")

        prediction\_result = predict\_financial\_metrics(request.user\_type, request.user\_data)

        return PredictionResponse(

            base\_interest\_rate=prediction\_result.get("base\_interest\_rate"),

            discounted\_interest\_rate=prediction\_result.get("discounted\_interest\_rate"),

            loan\_tenure=prediction\_result.get("loan\_tenure"),

            risk\_rating=prediction\_result.get("risk\_rating"),

            confidence\_score=prediction\_result.get("confidence\_score")

        )

    except Exception as e:

        logger.error(f" SELFESHTABLISHED loan assessment error: {e}")

        raise HTTPException(status\_code=500, detail=f"Assessment failed: {str(e)}")

@router.post("/selfeshtablished-client/feature-analysis", response\_model=ExplanationResponse)

async def analyze\_selfeshtablished\_client\_features(request: PredictionRequest):

    """

    Analyze feature importance for SELFESHTABLISHED client loan applications

    """

    try:

        if request.user\_type != "selfeshtablished":

            raise HTTPException(status\_code=400, detail="This endpoint is for SELFESHTABLISHED clients only")

        user\_type = request.user\_type

        features = prepare\_features(user\_type, request.user\_data)

        # SELFESHTABLISHED-specific explanation logic

        explanation = {}

        for feature, value in features.items():

            if feature == "Net\_Worth\_Cr":

                explanation[feature] = min(0.6, value / 100) if value > 0 else -0.3

            elif feature == "Investment\_Portfolio\_Diversity":

                explanation[feature] = min(0.4, value / 100) if value > 0 else -0.2

            elif feature == "Real\_Estate\_Value":

                explanation[feature] = min(0.3, value / 10000000) if value > 0 else -0.1

            elif feature == "Bank\_Relationship":

                explanation[feature] = min(0.25, value / 10) if value > 0 else -0.15

            elif feature == "Debt\_to\_Asset\_Ratio":

                explanation[feature] = -min(0.3, value) if value > 0.3 else 0.1

            elif feature == "Luxury\_Expenditure\_%":

                explanation[feature] = -min(0.2, value / 100) if value > 30 else 0.05

            else:

                explanation[feature] = 0.05 if value > 0 else -0.05

        # Generate predictions for recommendations

        prediction\_result = predict\_financial\_metrics(user\_type, request.user\_data)

        recommendations = generate\_recommendations(user\_type, features, prediction\_result)

        return ExplanationResponse(

            feature\_importance=explanation,

            recommendations=recommendations

        )

    except Exception as e:

        logger.error(f" SELFESHTABLISHED client feature analysis error: {e}")

        raise HTTPException(status\_code=500, detail=f"Feature analysis failed: {str(e)}")

@router.post("/selfeshtablished-client/generate-report")

async def generate\_selfeshtablished\_client\_report(request: ReportRequest):

    """

    Generate comprehensive PDF financial report for SELFESHTABLISHED clients

    """

    try:

        html\_content = generate\_enhanced\_html\_report(

            user\_profile=request.user\_profile,

            prediction=request.prediction,

            explanation=request.explanation,

            include\_charts=request.include\_charts,

            user\_role="SELFESHTABLISHED Client"

        )

        # PDF configuration

        options = {

            'page-size': 'A4',

            'margin-top': '0.75in',

            'margin-right': '0.75in',

            'margin-bottom': '0.75in',

            'margin-left': '0.75in',

            'encoding': "UTF-8",

            'no-outline': None,

            'enable-local-file-access': None

        }

        try:

            config = pdfkit.configuration(wkhtmltopdf="/usr/bin/wkhtmltopdf")

            pdf\_content = pdfkit.from\_string(html\_content, False, options=options, configuration=config)

        except:

            logger.warning("⚠️ wkhtmltopdf not found, using alternative PDF generation")

            pdf\_content = html\_content.encode('utf-8')

        filename = f"apexwolf\_selfeshtablished\_client\_report\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.pdf"

        return StreamingResponse(

            io.BytesIO(pdf\_content),

            media\_type="application/pdf",

            headers={"Content-Disposition": f"attachment; filename={filename}"}

        )

    except Exception as e:

        logger.error(f" SELFESHTABLISHED client report generation error: {e}")

        raise HTTPException(status\_code=500, detail=f"Report generation failed: {str(e)}")

# Emerging Entrepreneur Endpoints

@router.post("/emerging-entrepreneur/loan-assessment", response\_model=PredictionResponse)

async def assess\_emerging\_entrepreneur\_loan(request: PredictionRequest):

    """

    Assess loan terms for emerging entrepreneurs with specialized criteria

    """

    try:

        if request.user\_type != "emerging":

            raise HTTPException(status\_code=400, detail="This endpoint is for emerging entrepreneurs only")

        prediction\_result = predict\_financial\_metrics(request.user\_type, request.user\_data)

        return PredictionResponse(

            base\_interest\_rate=prediction\_result.get("base\_interest\_rate"),

            discounted\_interest\_rate=prediction\_result.get("discounted\_interest\_rate"),

            loan\_tenure=prediction\_result.get("loan\_tenure"),

            risk\_rating=prediction\_result.get("risk\_rating"),

            confidence\_score=prediction\_result.get("confidence\_score")

        )

    except Exception as e:

        logger.error(f" Emerging entrepreneur assessment error: {e}")

        raise HTTPException(status\_code=500, detail=f"Assessment failed: {str(e)}")

@router.post("/emerging-entrepreneur/feature-analysis", response\_model=ExplanationResponse)

async def analyze\_emerging\_entrepreneur\_features(request: PredictionRequest):

    """

    Analyze feature importance for emerging entrepreneur loan applications

    """

    try:

        if request.user\_type != "emerging":

            raise HTTPException(status\_code=400, detail="This endpoint is for emerging entrepreneurs only")

        user\_type = request.user\_type

        features = prepare\_features(user\_type, request.user\_data)

        # Emerging entrepreneur-specific explanation logic

        explanation = {}

        for feature, value in features.items():

            if feature == "Financial\_Literacy\_Score":

                explanation[feature] = min(0.4, value / 100) if value > 0 else -0.2

            elif feature == "Monthly\_Income":

                explanation[feature] = min(0.35, value / 100000) if value > 0 else -0.15

            elif feature == "Has\_Dependent\_Children":

                explanation[feature] = -0.1 if value == 1 else 0.05  # May affect risk assessment

            elif feature == "Employment\_Status":

                if value == 2:  # business

                    explanation[feature] = 0.2

                elif value == 1:  # self-employed

                    explanation[feature] = 0.1

                else:

                    explanation[feature] = 0.05

            elif feature == "Bank\_Affiliation":

                explanation[feature] = min(0.2, value / 10) if value > 0 else -0.1

            elif feature == "Repayment\_Behavior":

                explanation[feature] = min(0.3, value / 10) if value > 0 else -0.2

            else:

                explanation[feature] = 0.05 if value > 0 else -0.05

        # Generate predictions for recommendations

        prediction\_result = predict\_financial\_metrics(user\_type, request.user\_data)

        recommendations = generate\_recommendations(user\_type, features, prediction\_result)

        return ExplanationResponse(

            feature\_importance=explanation,

            recommendations=recommendations

        )

    except Exception as e:

        logger.error(f" Emerging entrepreneur feature analysis error: {e}")

        raise HTTPException(status\_code=500, detail=f"Feature analysis failed: {str(e)}")

@router.post("/emerging-entrepreneur/generate-report")

async def generate\_emerging\_entrepreneur\_report(request: ReportRequest):

    """

    Generate comprehensive PDF financial report for emerging entrepreneurs

    """

    try:

        html\_content = generate\_enhanced\_html\_report(

            user\_profile=request.user\_profile,

            prediction=request.prediction,

            explanation=request.explanation,

            include\_charts=request.include\_charts,

            user\_role="Emerging Entrepreneur"

        )

        # PDF configuration

        options = {

            'page-size': 'A4',

            'margin-top': '0.75in',

            'margin-right': '0.75in',

            'margin-bottom': '0.75in',

            'margin-left': '0.75in',

            'encoding': "UTF-8",

            'no-outline': None,

            'enable-local-file-access': None

        }

        try:

            config = pdfkit.configuration(wkhtmltopdf="/usr/bin/wkhtmltopdf")

            pdf\_content = pdfkit.from\_string(html\_content, False, options=options, configuration=config)

        except:

            logger.warning("⚠️ wkhtmltopdf not found, using alternative PDF generation")

            pdf\_content = html\_content.encode('utf-8')

        filename = f"apexwolf\_emerging\_entrepreneur\_report\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}.pdf"

        return StreamingResponse(

            io.BytesIO(pdf\_content),

            media\_type="application/pdf",

            headers={"Content-Disposition": f"attachment; filename={filename}"}

        )

    except Exception as e:

        logger.error(f" Emerging entrepreneur report generation error: {e}")

        raise HTTPException(status\_code=500, detail=f"Report generation failed: {str(e)}")

# Administrative Endpoints

@router.get("/admin/model-status")

async def get\_role\_based\_model\_status():

    """

    Get status and metadata of all loaded role-based models

    """

    status = {}

    for user\_type in USER\_TYPES:

        status[user\_type] = {}

        for model\_type in MODEL\_TYPES:

            model = user\_models.get(user\_type, {}).get(model\_type)

            metadata = model\_metadata.get(user\_type, {}).get(model\_type, {})

            status[user\_type][model\_type] = {

                "loaded": model is not None,

                "metadata": metadata

            }

    total\_loaded = sum(1 for ut in USER\_TYPES for mt in MODEL\_TYPES

                      if user\_models.get(ut, {}).get(mt) is not None)

    return {

        "timestamp": datetime.now(timezone.utc).isoformat(),

        "service": "ApexWolf Role-Based Financial Services",

        "models": status,

        "summary": {

            "total\_models": len(USER\_TYPES) \* len(MODEL\_TYPES),

            "loaded\_models": total\_loaded,

            "load\_percentage": round((total\_loaded / (len(USER\_TYPES) \* len(MODEL\_TYPES))) \* 100, 2)

        }

    }

@router.post("/admin/reload-models")

async def reload\_role\_based\_models(background\_tasks: BackgroundTasks):

    """

    Reload all role-based models (useful for model updates)

    """

    def reload\_task():

        try:

            load\_models()

            logger.info(" ApexWolf models reloaded successfully")

        except Exception as e:

            logger.error(f" ApexWolf model reload failed: {e}")

    background\_tasks.add\_task(reload\_task)

    return {

        "message": "ApexWolf role-based model reload initiated in background",

        "timestamp": datetime.now(timezone.utc).isoformat()

    }

# System Health Endpoint

@router.get("/system/health")

async def health\_check():

    """

    Health check endpoint for role-based financial services

    """

    # Check model availability

    total\_models = len(USER\_TYPES) \* len(MODEL\_TYPES)

    loaded\_models = sum(1 for ut in USER\_TYPES for mt in MODEL\_TYPES

                       if user\_models.get(ut, {}).get(mt) is not None)

    health\_status = "healthy" if loaded\_models > 0 else "degraded"

    return {

        "status": health\_status,

        "service": "ApexWolf Role-Based Financial Services",

        "timestamp": datetime.now(timezone.utc).isoformat(),

        "version": "2.0.0",

        "models": {

            "total": total\_models,

            "loaded": loaded\_models,

            "percentage": round((loaded\_models / total\_models) \* 100, 2)

        },

        "endpoints": {

            "structured\_employee": 3,

            "business\_owner": 3,

            "selfeshtablished\_client": 3,

            "emerging\_entrepreneur": 3,

            "admin": 2,

            "system": 1

        }

    }

# Generic endpoint for backwards compatibility (optional)

@router.post("/generic/loan-prediction", response\_model=PredictionResponse)

async def generic\_loan\_prediction(request: PredictionRequest):

    """

    Generic loan prediction endpoint (backwards compatibility)

    Use role-specific endpoints for better features

    """

    try:

        prediction\_result = predict\_financial\_metrics(request.user\_type, request.user\_data)

        return PredictionResponse(

            base\_interest\_rate=prediction\_result.get("base\_interest\_rate"),

            discounted\_interest\_rate=prediction\_result.get("discounted\_interest\_rate"),

            loan\_tenure=prediction\_result.get("loan\_tenure"),

            risk\_rating=prediction\_result.get("risk\_rating"),

            confidence\_score=prediction\_result.get("confidence\_score")

        )

    except Exception as e:

        logger.error(f" Generic prediction error: {e}")

        raise HTTPException(status\_code=500, detail=f"Prediction failed: {str(e)}")

# Welcome endpoint

@router.get("/")

async def welcome():

    """

    Welcome endpoint with API information

    """

    return {

        "service": "🐺 ApexWolf Role-Based Financial Services API",

        "version": "2.0.0",

        "description": "Advanced role-based financial assessment and loan prediction services",

        "roles": USER\_TYPES,

        "endpoints": {

            "structured\_employee": [

                "/structured-employee/loan-assessment",

                "/structured-employee/feature-analysis",

                "/structured-employee/generate-report"

            ],

            "business\_owner": [

                "/business-owner/loan-assessment",

                "/business-owner/feature-analysis",

                "/business-owner/generate-report"

            ],

            "selfeshtablished\_client": [

                "/selfeshtablished-client/loan-assessment",

                "/selfeshtablished-client/feature-analysis",

                "/selfeshtablished-client/generate-report"

            ],

            "emerging\_entrepreneur": [

                "/emerging-entrepreneur/loan-assessment",

                "/emerging-entrepreneur/feature-analysis",

                "/emerging-entrepreneur/generate-report"

            ],

            "admin": ["/admin/model-status", "/admin/reload-models"],

            "system": ["/system/health"]

        },

        "docs": "/docs",

        "timestamp": datetime.now(timezone.utc).isoformat()

    }

**INTEREST RATE OPTIMIZATION**

from fastapi import APIRouter, UploadFile, File, Form, HTTPException

from fastapi.responses import JSONResponse, FileResponse

from typing import Optional

import os

import shutil

from datetime import datetime

from fpdf import FPDF

from pydantic import BaseModel, field\_validator  # Pydantic V2 validator

from db import get\_connection

router = APIRouter()

UPLOAD\_DIR = "uploads"

os.makedirs(UPLOAD\_DIR, exist\_ok=True)

# Sentiment analysis keywords

GENUINE\_WORDS = [

    "medical", "surgery", "hospitalized", "emergency", "family issue", "accident", "laid off", "job loss",

    "bereavement", "funeral", "severe", "injury", "terminal", "pregnancy", "cancer", "heart", "ICU",

    "stroke", "hospital stay", "critical", "disability", "operation", "trauma", "treatment"

] + [f"{a} {b}" for a in ["medical", "emergency", "accident", "critical"] for b in ["issue", "bill", "case", "event"]]

MODERATE\_WORDS = [

    "financial stress", "temporary", "market", "inflation", "education", "child support", "tuition", "school fee",

    "moving", "rent increase", "repairs", "unexpected bills", "maintenance", "living cost", "college loan", "visa"

] + [f"{a} {b}" for a in ["education", "inflation", "expenses", "tuition"] for b in ["concern", "situation", "burden"]]

INVALID\_WORDS = [

    "just want", "prefer", "wish", "feel like", "because", "simply", "no reason", "nothing specific", "discount",

    "loyalty", "unfair", "not happy", "random", "better deal", "deserve"

] + [f"{a} {b}" for a in ["prefer", "deserve", "discount"] for b in ["option", "reason", "case"]]

UNCLEAR\_WORDS = [

    "help", "reduce", "urgent", "request", "assistance", "not sure", "please", "adjust", "change", "support",

    "review", "interest", "possible", "kindly", "feedback", "explain"

] + [f"{a} {b}" for a in ["help", "support", "adjust"] for b in ["case", "request", "situation", "concern"]]

# Analyze sentiment of the comment

def analyze\_sentiment(comment: str) -> str:

    comment\_lower = comment.lower()

    scores = {"Genuine": 0, "Moderate": 0, "Likely Invalid": 0, "Unclear or Not Valid": 0}

    for word in GENUINE\_WORDS:

        if word in comment\_lower:

            scores["Genuine"] += 1

    for word in MODERATE\_WORDS:

        if word in comment\_lower:

            scores["Moderate"] += 1

    for word in INVALID\_WORDS:

        if word in comment\_lower:

            scores["Likely Invalid"] += 1

    for word in UNCLEAR\_WORDS:

        if word in comment\_lower:

            scores["Unclear or Not Valid"] += 1

    return max(scores, key=scores.get) if any(scores.values()) else "Unclear or Not Valid"

# Pydantic model for admin response

class AdminResponse(BaseModel):

    id: int

    status: str

    admin\_note: str

    final\_rate: str

    @field\_validator("status")

    def valid\_status(cls, v):

        # Ensure status is approved or rejected

        if v not in ["approved", "rejected"]:

            raise ValueError("Status must be 'approved' or 'rejected'")

        return v

    @field\_validator("admin\_note")

    def valid\_note(cls, v):

        # Check admin note validity

        if not isinstance(v, str) or not v.strip():

            raise ValueError("Admin note cannot be empty")

        if len(v) > 500:

            raise ValueError("Admin note must not exceed 500 characters")

        return v

    @field\_validator("final\_rate")

    def valid\_rate(cls, v):

        # Check final rate is not empty

        if not isinstance(v, str) or not v.strip():

            raise ValueError("Final rate cannot be empty")

        return v

# Endpoint for submitting negotiation request

@router.post("/request")

async def submit\_negotiation(

    loan\_type: str = Form(...),

    loan\_amount: str = Form(...),

    current\_rate: str = Form(...),

    expected\_rate: str = Form(...),

    comment: str = Form(...),

    file: Optional[UploadFile] = File(None)

):

    # Validate fields

    if not loan\_type or not loan\_type.strip():

        raise HTTPException(status\_code=400, detail="Loan type is required")

    if not loan\_amount or not loan\_amount.strip() or not loan\_amount.replace('.', '', 1).isdigit() or float(loan\_amount) <= 0:

        raise HTTPException(status\_code=400, detail="Loan amount must be a positive number")

    if not current\_rate or not current\_rate.strip():

        raise HTTPException(status\_code=400, detail="Current rate is required")

    if not expected\_rate or not expected\_rate.strip():

        raise HTTPException(status\_code=400, detail="Expected rate is required")

    if not comment or not comment.strip():

        raise HTTPException(status\_code=400, detail="Comment is required")

    if len(comment) > 1000:

        raise HTTPException(status\_code=400, detail="Comment must not exceed 1000 characters")

    # Save uploaded file if present

    file\_path = ""

    if file:

        filename = f"{datetime.now().strftime('%Y%m%d%H%M%S')}\_{file.filename}"

        file\_path = os.path.join(UPLOAD\_DIR, filename)

        with open(file\_path, "wb") as buffer:

            shutil.copyfileobj(file.file, buffer)

        file\_path = filename

    # Analyze comment sentiment

    sentiment = analyze\_sentiment(comment)

    conn = get\_connection()

    try:

        with conn.cursor() as cursor:

            cursor.execute(

                "INSERT INTO negotiation\_requests "

                "(loan\_type, loan\_amount, current\_rate, expected\_rate, comment, file\_path, status, admin\_note, sentiment, final\_rate, timestamp) "

                "VALUES (%s, %s, %s, %s, %s, %s, 'pending', '', %s, '', %s)",

                (loan\_type, loan\_amount, current\_rate, expected\_rate, comment, file\_path, sentiment, datetime.now())

            )

            conn.commit()

    finally:

        conn.close()

    return JSONResponse(content={"msg": "Negotiation request submitted successfully.", "sentiment": sentiment})

# Endpoint for user to get latest negotiation status

@router.get("/status")

def get\_user\_latest():

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("SELECT \* FROM negotiation\_requests ORDER BY id DESC")

            records = cursor.fetchall()

            if not records:

                return {"status": "none"}

            for record in records:

                if record["status"] in ["approved", "rejected"]:

                    return record

            return records[0]

    finally:

        conn.close()

# Endpoint for admin to get all pending requests

@router.get("/admin/pending")

def get\_pending():

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("SELECT \* FROM negotiation\_requests WHERE status = 'pending'")

            return {"requests": cursor.fetchall()}

    finally:

        conn.close()

# Endpoint for admin to get all responded requests

@router.get("/admin/responded")

def get\_responded():

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("SELECT \* FROM negotiation\_requests WHERE status IN ('approved', 'rejected')")

            return {"requests": cursor.fetchall()}

    finally:

        conn.close()

# Endpoint for admin to respond to a negotiation

@router.post("/admin/respond")

def respond\_admin(response: AdminResponse):

    conn = get\_connection()

    try:

        with conn.cursor() as cursor:

            cursor.execute(

                "UPDATE negotiation\_requests SET status = %s, admin\_note = %s, final\_rate = %s "

                "WHERE id = %s AND status = 'pending'",

                (response.status, response.admin\_note, response.final\_rate, response.id)

            )

            conn.commit()

            if cursor.rowcount == 0:

                raise HTTPException(status\_code=404, detail="Pending request not found")

    finally:

        conn.close()

    return {"msg": "Response updated successfully."}

# Custom PDF class for receipts

class ReceiptPDF(FPDF):

    def header(self):

        # PDF header

        self.set\_font("Arial", "B", 14)

        self.set\_text\_color(40, 40, 120)

        self.cell(0, 10, "ApexWolf AI Finance", ln=True, align="C")

        self.ln(5)

        self.set\_draw\_color(180, 180, 180)

        self.line(10, 25, 200, 25)

        self.ln(10)

    def footer(self):

        # PDF footer

        self.set\_y(-20)

        self.set\_draw\_color(200, 200, 200)

        self.line(10, self.get\_y(), 200, self.get\_y())

        self.set\_font("Arial", "I", 9)

        self.set\_text\_color(100, 100, 100)

        self.cell(0, 10, "Generated by ApexWolf | Confidential", 0, 0, "C")

# Endpoint to generate PDF receipt for a negotiation

@router.get("/receipt/{id}")

def generate\_receipt(id: int):

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("SELECT \* FROM negotiation\_requests WHERE id = %s", (id,))

            req = cursor.fetchone()

            if not req:

                raise HTTPException(status\_code=404, detail="Request not found")

    finally:

        conn.close()

    pdf = ReceiptPDF()

    pdf.add\_page()

    pdf.set\_font("Arial", size=12)

    pdf.set\_text\_color(0, 0, 0)

    pdf.cell(0, 10, txt="Rate Optimization Summary", ln=True)

    pdf.ln(5)

    pdf.set\_fill\_color(240, 240, 240)

    pdf.set\_font("Arial", "B", 12)

    pdf.cell(60, 10, "Field", 1, 0, "C", 1)

    pdf.cell(130, 10, "Value", 1, 1, "C", 1)

    fields = ["loan\_type", "loan\_amount", "current\_rate", "expected\_rate", "comment",

              "sentiment", "status", "final\_rate", "admin\_note"]

    pdf.set\_font("Arial", "", 12)

    for key in fields:

        label = key.replace('\_', ' ').title()

        value = str(req.get(key, ''))

        if len(value) > 80:

            value = value[:77] + "..."

        pdf.cell(60, 10, label, 1)

        pdf.cell(130, 10, value, 1)

        pdf.ln()

    filename = f"receipt\_{id}.pdf"

    filepath = os.path.join(UPLOAD\_DIR, filename)

    pdf.output(filepath)

    return FileResponse(filepath, media\_type='application/pdf', filename=filename)

**SMART INVESTMENT TRACKER**

from fastapi import APIRouter, Request

from fastapi.responses import JSONResponse

from datetime import datetime

from dateutil.relativedelta import relativedelta

import calendar

import os

import json

router = APIRouter()

NAV\_FILE = "nav\_data.json"

INTEREST\_RATE\_FILE = "interest\_rate.json"

# Load NAV data from the JSON file; return empty dict if file does not exist

def load\_nav\_data():

    if not os.path.exists(NAV\_FILE):

        return {}

    with open(NAV\_FILE, "r") as f:

        return json.load(f)

# Save NAV data to the JSON file

def save\_nav\_data(data):

    with open(NAV\_FILE, "w") as f:

        json.dump(data, f, indent=2)

# Load the default interest rate from a JSON file

def load\_interest\_rate():

    if not os.path.exists(INTEREST\_RATE\_FILE):

        return None

    with open(INTEREST\_RATE\_FILE, "r") as f:

        data = json.load(f)

        return data.get("interest\_rate")

# Save the default interest rate to a JSON file

def save\_interest\_rate(rate):

    with open(INTEREST\_RATE\_FILE, "w") as f:

        json.dump({"interest\_rate": rate}, f, indent=2)

# Calculate future value for prediction mode (compound interest + SIP)

def calculate\_prediction(amount, rate, duration, sip=0):

    rate\_per\_month = (rate / 100) / 12

    future\_value = amount \* ((1 + rate\_per\_month) \*\* duration)

    for i in range(duration):

        future\_value += sip \* ((1 + rate\_per\_month) \*\* (duration - i - 1))

    return round(future\_value, 2)

# Endpoint: POST /track

# Predicts future investment value or tracks growth using NAV data

@router.post("/track")

async def process\_investment(request: Request):

    try:

        data = await request.json()

        mode = data.get("mode")

        amount = float(data.get("amount", 0))

        duration = int(data.get("duration", 0))

    except Exception:

        return JSONResponse({"error": "Invalid input structure or values"}, status\_code=400)

    if amount <= 0 or duration <= 0:

        return JSONResponse({"error": "Amount and duration must be positive"}, status\_code=400)

    months = []

    values = []

    # Prediction mode

    if mode == "predict":

        # Use user-provided interest\_rate if present, else use stored default

        interest\_rate = data.get("interest\_rate")

        if interest\_rate is None:

            interest\_rate = load\_interest\_rate()

            if interest\_rate is None:

                return JSONResponse({"error": "Interest rate must be provided or set by admin"}, status\_code=400)

        try:

            rate = float(interest\_rate)

            sip = float(data.get("sip", 0))

        except Exception:

            return JSONResponse({"error": "Invalid interest rate or SIP"}, status\_code=400)

        now = datetime.now()

        for i in range(duration):

            future\_date = now + relativedelta(months=i)

            month\_name = calendar.month\_abbr[future\_date.month]

            label = f"{month\_name} {future\_date.year}"

            value = calculate\_prediction(amount, rate, i + 1, sip)

            months.append(label)

            values.append(round(value, 2))

        return {"months": months, "values": values}

    # Tracking mode

    elif mode == "track":

        start\_month = data.get("start\_month")

        try:

            start\_dt = datetime.strptime(start\_month, "%Y-%m")

        except:

            return JSONResponse({"error": "Invalid start month format (use YYYY-MM)"}, status\_code=400)

        navs = load\_nav\_data()

        start\_key = calendar.month\_abbr[start\_dt.month]

        start\_nav = float(navs.get(start\_key, 12.0))

        nav\_list = []

        for i in range(duration):

            current\_date = start\_dt + relativedelta(months=i)

            key = calendar.month\_abbr[current\_date.month]

            current\_nav = float(navs.get(key, start\_nav + (i \* 0.4)))

            label = f"{key} {current\_date.year}"

            value = round(amount \* (current\_nav / start\_nav), 2)

            months.append(label)

            values.append(value)

            nav\_list.append(current\_nav)

        return {"months": months, "values": values, "navs": nav\_list}

    return JSONResponse({"error": "Invalid mode (use 'predict' or 'track')"}, status\_code=400)

# Endpoint: POST /nav

# Save or update NAV value for a specific month (admin use)

@router.post("/nav")

async def save\_nav(request: Request):

    try:

        data = await request.json()

        month = data.get("month")

        nav = float(data.get("nav"))

    except Exception:

        return JSONResponse({"error": "Invalid input"}, status\_code=400)

    # Validate month abbreviation

    if not month or not month.isalpha() or len(month) != 3:

        return JSONResponse({"error": "Invalid month abbreviation (e.g. 'Jan')"}, status\_code=400)

    navs = load\_nav\_data()

    navs[month.capitalize()] = nav

    save\_nav\_data(navs)

    return {"status": "ok", "month": month.capitalize(), "nav": nav}

# Endpoint: GET /navs

# Return all stored NAV values (admin use)

@router.get("/navs")

async def get\_all\_navs():

    navs = load\_nav\_data()

    return [{"month": k, "nav": v} for k, v in navs.items()]

# Endpoint: POST /interest\_rate

# Set or update the default interest rate (admin use)

@router.post("/interest\_rate")

async def set\_interest\_rate(request: Request):

    try:

        data = await request.json()

        interest\_rate = float(data.get("interest\_rate"))

    except Exception:

        return JSONResponse({"error": "Invalid input"}, status\_code=400)

    if interest\_rate <= 0:

        return JSONResponse({"error": "Interest rate must be a positive number"}, status\_code=400)

    save\_interest\_rate(interest\_rate)

    return {"status": "ok", "interest\_rate": interest\_rate}

# Endpoint: GET /interest\_rate

# Get the current default interest rate (admin use)

@router.get("/interest\_rate")

async def get\_interest\_rate():

    rate = load\_interest\_rate()

    if rate is None:

        return JSONResponse({"error": "Interest rate not set"}, status\_code=404)

    return {"interest\_rate": rate}

# Export router to be included in main.py

investment\_router = router

**MARKET TRENDS**

from fastapi import APIRouter, HTTPException

from pydantic import BaseModel

from datetime import datetime

from typing import List

import json

import os

import re

import aiohttp

import asyncio

import yfinance as yf

# Create FastAPI router for market trends endpoints

router = APIRouter()

CACHE\_DURATION = 300  # Cache duration in seconds (5 minutes)

ADMIN\_FILE = "admin\_rates.json"  # File to persist admin-set rates

CACHE\_FILE = "market\_cache.json"  # File to cache latest response

# Pydantic model for a single bank interest rate

class InterestRate(BaseModel):

    bank: str

    rate: str

    source: str  # "Auto" or "Manual"

# Pydantic model for a single market index value

class MarketIndicator(BaseModel):

    category: str

    symbol: str

    last: float

    currency: str

# Pydantic model for full API response

class MarketTrendsResponse(BaseModel):

    timestamp: str

    interest\_rates: List[InterestRate]

    market\_trends: List[MarketIndicator]

# Dictionary of banks with scraping URLs and regex for rate extraction

BANK\_SOURCES = {

    "State Bank of India": {

        "url": "https://sbi.co.in/web/interest-rates/deposit-rates/savings-bank-deposit",

        "regex": r"(\d+\.\d+)%"

    },

    "HDFC Bank": {

        "url": "https://www.hdfcbank.com/personal/save/accounts/savings-accounts/regular-savings-account",

        "regex": r"(\d+\.\d+)%"

    },

    "ICICI Bank": {

        "url": "https://www.icicibank.com/personal-banking/accounts/savings-account",

        "regex": r"(\d+\.\d+)%"

    },

    "Axis Bank": {

        "url": "https://www.axisbank.com/personal/accounts/savings-account/basic-savings-account",

        "regex": r"(\d+\.\d+)%"

    },

    "Kotak Mahindra Bank": {

        "url": "https://www.kotak.com/en/personal-banking/accounts/savings-account.html",

        "regex": r"(\d+\.\d+)%"

    }

}

# Dictionary of major market indices to track with yfinance symbol and currency

MARKET\_SYMBOLS = {

    "Nifty 50": {"symbol": "^NSEI", "currency": "INR"},

    "Sensex": {"symbol": "^BSESN", "currency": "INR"},

    "Dow Jones": {"symbol": "^DJI", "currency": "USD"},

    "Nasdaq": {"symbol": "^IXIC", "currency": "USD"},

    "S&P 500": {"symbol": "^GSPC", "currency": "USD"}

}

# Load admin-set rates from file

def load\_admin\_rates():

    if not os.path.exists(ADMIN\_FILE):

        return {}

    with open(ADMIN\_FILE, "r") as f:

        try:

            return json.load(f)

        except:

            return {}

# Save admin-set rate for a bank

def save\_admin\_rate(bank: str, rate: str):

    rates = load\_admin\_rates()

    rates[bank] = rate

    with open(ADMIN\_FILE, "w") as f:

        json.dump(rates, f)

# Load cached market trends response from file

def load\_cache():

    if not os.path.exists(CACHE\_FILE):

        return {}

    with open(CACHE\_FILE, "r") as f:

        try:

            return json.load(f)

        except:

            return {}

# Save cache to file

def save\_cache(data: dict):

    with open(CACHE\_FILE, "w") as f:

        json.dump(data, f)

# Async function to scrape a bank's rate using aiohttp and regex

async def fetch\_bank\_rate(bank, url, regex):

    try:

        async with aiohttp.ClientSession() as session:

            async with session.get(url, timeout=10) as resp:

                html = await resp.text()

                match = re.search(regex, html)

                if match:

                    return {"bank": bank, "rate": f"{match.group(1)}%", "source": "Auto"}

    except:

        pass

    return {"bank": bank, "rate": "Error", "source": "Auto"}

# Fetch live index data using yfinance

def fetch\_market\_data():

    result = []

    for name, info in MARKET\_SYMBOLS.items():

        try:

            ticker = yf.Ticker(info["symbol"])

            data = ticker.history(period="1d")

            last\_price = float(data["Close"].iloc[-1]) if not data.empty else 0.0

            result.append({

                "category": name,

                "symbol": info["symbol"],

                "last": round(last\_price, 2),

                "currency": info["currency"]

            })

        except:

            result.append({

                "category": name,

                "symbol": info["symbol"],

                "last": 0.0,

                "currency": info["currency"]

            })

    return result

# Endpoint: GET /markettrends

# Returns interest rates (auto/manual) and current market indices

@router.get("/markettrends", response\_model=MarketTrendsResponse)

async def get\_market\_trends():

    now = datetime.now().isoformat()

    cache = load\_cache()

    if "timestamp" in cache:

        last = datetime.fromisoformat(cache["timestamp"])

        if (datetime.now() - last).total\_seconds() < CACHE\_DURATION:

            return cache

    admin\_rates = load\_admin\_rates()

    tasks = []

    for bank, info in BANK\_SOURCES.items():

        tasks.append(fetch\_bank\_rate(bank, info["url"], info["regex"]))

    auto\_rates = await asyncio.gather(\*tasks)

    final\_rates = []

    for rate in auto\_rates:

        # Use admin rate as fallback if scraping fails

        if rate["rate"] == "Error" and rate["bank"] in admin\_rates:

            final\_rates.append({

                "bank": rate["bank"],

                "rate": admin\_rates[rate["bank"]],

                "source": "Manual"

            })

        else:

            final\_rates.append(rate)

    market\_data = fetch\_market\_data()

    response = {

        "timestamp": now,

        "interest\_rates": final\_rates,

        "market\_trends": market\_data

    }

    save\_cache(response)

    return response

# Endpoint: POST /markettrends/admin/update

# Allows admin to set/override a bank's interest rate as fallback

@router.post("/markettrends/admin/update")

def update\_admin\_rate(rate: InterestRate):

    save\_admin\_rate(rate.bank, rate.rate)

    return {"message": f"{rate.bank} rate updated to {rate.rate} manually."}

**REPO RATE**

from fastapi import APIRouter, HTTPException, BackgroundTasks

from pydantic import BaseModel

from typing import List, Optional, Dict

import requests

from bs4 import BeautifulSoup

import json

from db import get\_connection

from datetime import datetime, timedelta

import re

import logging

from urllib.parse import urljoin

import asyncio

import aiohttp

from contextlib import asynccontextmanager

import mysql.connector

from mysql.connector import Error

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

router = APIRouter(prefix="/api/repo-rate", tags=["Repo Rate"])

# Pydantic models

class RepoRateData(BaseModel):

    month: str

    repo\_rate: float

    reverse\_repo\_rate: float

    effective\_date: str

    month\_number: int

class RepoRateResponse(BaseModel):

    year: int

    data: List[RepoRateData]

    last\_updated: str

    source: str

    total\_changes: int

class RepoRateHistory(BaseModel):

    available\_years: List[int]

    current\_repo\_rate: float

    current\_reverse\_repo\_rate: float

    last\_change\_date: str

# Database setup

def init\_db():

    """Initialize MySQL database for repo rate data"""

    try:

        conn = get\_connection()

        cursor = conn.cursor()

        cursor.execute('''

            CREATE TABLE IF NOT EXISTS repo\_rates (

                id INT AUTO\_INCREMENT PRIMARY KEY,

                year INT NOT NULL,

                month INT NOT NULL,

                month\_name VARCHAR(10) NOT NULL,

                repo\_rate DECIMAL(5,2) NOT NULL,

                reverse\_repo\_rate DECIMAL(5,2) NOT NULL,

                effective\_date DATE NOT NULL,

                created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

                UNIQUE KEY unique\_rate (year, month, effective\_date),

                INDEX idx\_year (year),

                INDEX idx\_date (effective\_date)

            ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

        ''')

        cursor.execute('''

            CREATE TABLE IF NOT EXISTS repo\_rate\_update\_log (

                id INT AUTO\_INCREMENT PRIMARY KEY,

                last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

                source\_url TEXT,

                records\_updated INT DEFAULT 0,

                status VARCHAR(50) DEFAULT 'SUCCESS',

                error\_message TEXT,

                INDEX idx\_updated (last\_updated)

            ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4

        ''')

        conn.commit()

        cursor.close()

        conn.close()

        logger.info("Database tables initialized successfully")

    except Error as e:

        logger.error(f"Error initializing database: {str(e)}")

        raise

# Initialize database on startup

init\_db()

class RBIDataFetcher:

    """Class to handle RBI data fetching and parsing"""

    def \_\_init\_\_(self):

        self.rbi\_urls = [

            "https://www.rbi.org.in/Scripts/BS\_PressReleaseDisplay.aspx?prid=54839",  # Latest monetary policy

            "https://www.rbi.org.in/Scripts/MonetaryPolicyRates.aspx",  # Current rates page

            "https://rbi.org.in/Scripts/WSSView.aspx?Id=240",  # Historical repo rate data

            "https://www.rbi.org.in/Scripts/BS\_ViewBulletin.aspx?Id=21002"  # RBI bulletin

        ]

        self.headers = {

            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',

            'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,\*/\*;q=0.8',

            'Accept-Language': 'en-US,en;q=0.9',

            'Accept-Encoding': 'gzip, deflate, br',

            'Connection': 'keep-alive',

            'Upgrade-Insecure-Requests': '1',

            'Sec-Fetch-Dest': 'document',

            'Sec-Fetch-Mode': 'navigate',

            'Sec-Fetch-Site': 'none',

            'Sec-Fetch-User': '?1'

        }

    async def fetch\_current\_rates(self) -> Dict:

        """Fetch current repo rates from RBI website"""

        try:

            async with aiohttp.ClientSession(headers=self.headers) as session:

                # Try to fetch from RBI monetary policy page

                for url in self.rbi\_urls:

                    try:

                        async with session.get(url, timeout=30) as response:

                            if response.status == 200:

                                html = await response.text()

                                rates = self.\_parse\_repo\_rates(html)

                                if rates:

                                    return rates

                    except Exception as e:

                        logger.warning(f"Failed to fetch from {url}: {str(e)}")

                        continue

                # Fallback: Use RBI's official API endpoint if available

                return await self.\_fetch\_from\_rbi\_api()

        except Exception as e:

            logger.error(f"Error fetching current rates: {str(e)}")

            return self.\_get\_fallback\_rates()

    def \_parse\_repo\_rates(self, html: str) -> Optional[Dict]:

        """Parse repo rates from RBI HTML content"""

        try:

            soup = BeautifulSoup(html, 'html.parser')

            # Look for repo rate patterns in text

            text = soup.get\_text().lower()

            # Enhanced regex patterns to find repo rates

            repo\_patterns = [

                r'repo\s\*rate[:\s]\*(\d+\.?\d\*)\s\*%?',

                r'policy\s\*repo\s\*rate[:\s]\*(\d+\.?\d\*)\s\*%?',

                r'benchmark\s\*rate[:\s]\*(\d+\.?\d\*)\s\*%?'

            ]

            reverse\_repo\_patterns = [

                r'reverse\s\*repo\s\*rate[:\s]\*(\d+\.?\d\*)\s\*%?',

                r'standing\s\*deposit\s\*facility[:\s]\*(\d+\.?\d\*)\s\*%?'

            ]

            repo\_rate = None

            reverse\_repo\_rate = None

            # Try multiple patterns for repo rate

            for pattern in repo\_patterns:

                match = re.search(pattern, text)

                if match:

                    repo\_rate = float(match.group(1))

                    break

            # Try multiple patterns for reverse repo rate

            for pattern in reverse\_repo\_patterns:

                match = re.search(pattern, text)

                if match:

                    reverse\_repo\_rate = float(match.group(1))

                    break

            # If we found repo rate, use it

            if repo\_rate:

                # If reverse repo rate not found, use typical spread

                if not reverse\_repo\_rate:

                    reverse\_repo\_rate = 3.35  # Current reverse repo rate

                return {

                    'repo\_rate': repo\_rate,

                    'reverse\_repo\_rate': reverse\_repo\_rate,

                    'effective\_date': datetime.now().strftime('%Y-%m-%d'),

                    'source': 'RBI Official Website - Live Data'

                }

            # Look for tables with rate information

            tables = soup.find\_all('table')

            for table in tables:

                rows = table.find\_all('tr')

                for row in rows:

                    cells = row.find\_all(['td', 'th'])

                    if len(cells) >= 2:

                        cell\_text = ' '.join([cell.get\_text().strip().lower() for cell in cells])

                        if 'repo rate' in cell\_text or 'policy rate' in cell\_text:

                            # Extract rate from the row

                            rate\_match = re.search(r'(\d+\.?\d\*)', cell\_text)

                            if rate\_match:

                                repo\_rate = float(rate\_match.group(1))

                        if 'reverse repo' in cell\_text:

                            rate\_match = re.search(r'(\d+\.?\d\*)', cell\_text)

                            if rate\_match:

                                reverse\_repo\_rate = float(rate\_match.group(1))

                if repo\_rate:

                    return {

                        'repo\_rate': repo\_rate,

                        'reverse\_repo\_rate': reverse\_repo\_rate or 3.35,

                        'effective\_date': datetime.now().strftime('%Y-%m-%d'),

                        'source': 'RBI Official Website - Table Data'

                    }

        except Exception as e:

            logger.error(f"Error parsing repo rates: {str(e)}")

        return None

    async def \_fetch\_from\_rbi\_api(self) -> Dict:

        """Attempt to fetch from RBI's data API if available"""

        try:

            # RBI sometimes provides JSON data endpoints

            api\_urls = [

                "https://www.rbi.org.in/Scripts/bs\_viewcontent.aspx?Id=2009",

                "https://rbi.org.in/Scripts/MonetaryPolicyRates.aspx"

            ]

            async with aiohttp.ClientSession() as session:

                for url in api\_urls:

                    try:

                        async with session.get(url, headers=self.headers) as response:

                            if response.status == 200:

                                content = await response.text()

                                rates = self.\_parse\_repo\_rates(content)

                                if rates:

                                    return rates

                    except:

                        continue

        except Exception as e:

            logger.error(f"API fetch failed: {str(e)}")

        return self.\_get\_fallback\_rates()

    def \_get\_fallback\_rates(self) -> Dict:

        """Fallback rates based on last known RBI rates"""

        return {

            'repo\_rate': 5.50,  # Current as of June 6, 2025

            'reverse\_repo\_rate': 3.35,

            'effective\_date': datetime.now().strftime('%Y-%m-%d'),

            'source': 'RBI Official Data (Last Known Rates as of June 6, 2025)'

        }

    def generate\_monthly\_data(self, year: int, current\_rates: Dict) -> List[Dict]:

        """Generate monthly repo rate data for a given year based on actual RBI rate changes"""

        months = [

            'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',

            'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'

        ]

        # Real historical rate changes based on RBI announcements

        rate\_changes\_by\_year = {

            2020: {

                1: {'repo': 5.15, 'reverse': 3.35},  # January 2020

                2: {'repo': 5.15, 'reverse': 3.35},  # February (before COVID cuts)

                3: {'repo': 4.40, 'reverse': 3.35},  # March 27, 2020 - Emergency cut

                4: {'repo': 4.40, 'reverse': 3.35},  # April

                5: {'repo': 4.00, 'reverse': 3.35},  # May 22, 2020 - Further cut

                6: {'repo': 4.00, 'reverse': 3.35},

                7: {'repo': 4.00, 'reverse': 3.35},

                8: {'repo': 4.00, 'reverse': 3.35},

                9: {'repo': 4.00, 'reverse': 3.35},

                10: {'repo': 4.00, 'reverse': 3.35},

                11: {'repo': 4.00, 'reverse': 3.35},

                12: {'repo': 4.00, 'reverse': 3.35}

            },

            2021: {

                1: {'repo': 4.00, 'reverse': 3.35},

                2: {'repo': 4.00, 'reverse': 3.35},

                3: {'repo': 4.00, 'reverse': 3.35},

                4: {'repo': 4.00, 'reverse': 3.35},

                5: {'repo': 4.00, 'reverse': 3.35},

                6: {'repo': 4.00, 'reverse': 3.35},

                7: {'repo': 4.00, 'reverse': 3.35},

                8: {'repo': 4.00, 'reverse': 3.35},

                9: {'repo': 4.00, 'reverse': 3.35},

                10: {'repo': 4.00, 'reverse': 3.35},

                11: {'repo': 4.00, 'reverse': 3.35},

                12: {'repo': 4.00, 'reverse': 3.35}

            },

            2022: {

                1: {'repo': 4.00, 'reverse': 3.35},

                2: {'repo': 4.00, 'reverse': 3.35},

                3: {'repo': 4.00, 'reverse': 3.35},

                4: {'repo': 4.00, 'reverse': 3.35},

                5: {'repo': 4.40, 'reverse': 3.35},  # May 4, 2022 - First hike

                6: {'repo': 4.90, 'reverse': 3.35},  # June 8, 2022

                7: {'repo': 4.90, 'reverse': 3.35},

                8: {'repo': 5.40, 'reverse': 3.35},  # August 5, 2022

                9: {'repo': 5.90, 'reverse': 3.35},  # September 30, 2022

                10: {'repo': 5.90, 'reverse': 3.35},

                11: {'repo': 5.90, 'reverse': 3.35},

                12: {'repo': 6.25, 'reverse': 3.35}  # December 7, 2022

            },

            2023: {

                1: {'repo': 6.25, 'reverse': 3.35},

                2: {'repo': 6.50, 'reverse': 3.35},  # February 8, 2023

                3: {'repo': 6.50, 'reverse': 3.35},

                4: {'repo': 6.50, 'reverse': 3.35},  # April - unchanged

                5: {'repo': 6.50, 'reverse': 3.35},

                6: {'repo': 6.50, 'reverse': 3.35},  # June - unchanged

                7: {'repo': 6.50, 'reverse': 3.35},

                8: {'repo': 6.50, 'reverse': 3.35},  # August - unchanged

                9: {'repo': 6.50, 'reverse': 3.35},

                10: {'repo': 6.50, 'reverse': 3.35}, # October - unchanged

                11: {'repo': 6.50, 'reverse': 3.35},

                12: {'repo': 6.50, 'reverse': 3.35}  # December - unchanged

            },

            2024: {

                1: {'repo': 6.50, 'reverse': 3.35},

                2: {'repo': 6.50, 'reverse': 3.35},  # February 8, 2024 - unchanged

                3: {'repo': 6.50, 'reverse': 3.35},

                4: {'repo': 6.50, 'reverse': 3.35},  # April - unchanged

                5: {'repo': 6.50, 'reverse': 3.35},

                6: {'repo': 6.50, 'reverse': 3.35},  # June - unchanged

                7: {'repo': 6.50, 'reverse': 3.35},

                8: {'repo': 6.50, 'reverse': 3.35},  # August - unchanged

                9: {'repo': 6.50, 'reverse': 3.35},

                10: {'repo': 6.50, 'reverse': 3.35}, # October - unchanged

                11: {'repo': 6.50, 'reverse': 3.35},

                12: {'repo': 6.50, 'reverse': 3.35}  # December - unchanged

            },

            2025: {

                1: {'repo': 6.50, 'reverse': 3.35},

                2: {'repo': 6.25, 'reverse': 3.35},  # February cut

                3: {'repo': 6.25, 'reverse': 3.35},

                4: {'repo': 6.00, 'reverse': 3.35},  # April cut

                5: {'repo': 6.00, 'reverse': 3.35},

                6: {'repo': 5.50, 'reverse': 3.35},  # June 6, 2025 - 50 bps cut

                7: {'repo': 5.50, 'reverse': 3.35},

                8: {'repo': 5.50, 'reverse': 3.35},

                9: {'repo': 5.50, 'reverse': 3.35},

                10: {'repo': 5.50, 'reverse': 3.35},

                11: {'repo': 5.50, 'reverse': 3.35},

                12: {'repo': 5.50, 'reverse': 3.35}

            }

        }

        monthly\_data = []

        # Get year-specific rates or use current rates as fallback

        year\_rates = rate\_changes\_by\_year.get(year, {})

        for i, month in enumerate(months, 1):

            # Use historical data if available, otherwise use current rates

            if year\_rates and i in year\_rates:

                repo\_rate = year\_rates[i]['repo']

                reverse\_repo\_rate = year\_rates[i]['reverse']

            else:

                # Fallback to current rates for future months or missing data

                repo\_rate = current\_rates.get('repo\_rate', 5.50)

                reverse\_repo\_rate = current\_rates.get('reverse\_repo\_rate', 3.35)

            monthly\_data.append({

                'month': month,

                'month\_number': i,

                'repo\_rate': repo\_rate,

                'reverse\_repo\_rate': reverse\_repo\_rate,

                'effective\_date': f'{year}-{i:02d}-01'

            })

        return monthly\_data

# Initialize data fetcher

rbi\_fetcher = RBIDataFetcher()

@router.get("/current", response\_model=Dict)

async def get\_current\_repo\_rate():

    """Get current repo rate from RBI"""

    try:

        current\_rates = await rbi\_fetcher.fetch\_current\_rates()

        # Update database with current rates

        conn = get\_connection()

        cursor = conn.cursor()

        now = datetime.now()

        cursor.execute('''

            INSERT INTO repo\_rates

            (year, month, month\_name, repo\_rate, reverse\_repo\_rate, effective\_date)

            VALUES (%s, %s, %s, %s, %s, %s)

            ON DUPLICATE KEY UPDATE

            repo\_rate = VALUES(repo\_rate),

            reverse\_repo\_rate = VALUES(reverse\_repo\_rate),

            created\_at = CURRENT\_TIMESTAMP

        ''', (

            now.year, now.month, now.strftime('%b'),

            current\_rates['repo\_rate'], current\_rates['reverse\_repo\_rate'],

            current\_rates['effective\_date']

        ))

        conn.commit()

        cursor.close()

        conn.close()

        return {

            "current\_repo\_rate": current\_rates['repo\_rate'],

            "current\_reverse\_repo\_rate": current\_rates['reverse\_repo\_rate'],

            "effective\_date": current\_rates['effective\_date'],

            "source": current\_rates['source'],

            "last\_updated": datetime.now().isoformat()

        }

    except Error as e:

        logger.error(f"Database error getting current repo rate: {str(e)}")

        raise HTTPException(status\_code=500, detail=f"Database error: {str(e)}")

    except Exception as e:

        logger.error(f"Error getting current repo rate: {str(e)}")

        raise HTTPException(status\_code=500, detail=f"Failed to fetch current repo rate: {str(e)}")

@router.get("/history", response\_model=RepoRateHistory)

async def get\_repo\_rate\_history():

    """Get available years and current rate info"""

    try:

        current\_rates = await rbi\_fetcher.fetch\_current\_rates()

        return RepoRateHistory(

            available\_years=[2020, 2021, 2022, 2023, 2024, 2025],

            current\_repo\_rate=current\_rates['repo\_rate'],

            current\_reverse\_repo\_rate=current\_rates['reverse\_repo\_rate'],

            last\_change\_date=current\_rates['effective\_date']

        )

    except Exception as e:

        logger.error(f"Error getting repo rate history: {str(e)}")

        raise HTTPException(status\_code=500, detail=f"Failed to fetch repo rate history: {str(e)}")

@router.get("/{year}", response\_model=RepoRateResponse)

async def get\_repo\_rate\_by\_year(year: int):

    """Get repo rate data for a specific year"""

    try:

        if year < 2020 or year > 2025:

            raise HTTPException(status\_code=400, detail="Year must be between 2020 and 2025")

        # Get current rates from RBI

        current\_rates = await rbi\_fetcher.fetch\_current\_rates()

        # Generate monthly data for the year

        monthly\_data = rbi\_fetcher.generate\_monthly\_data(year, current\_rates)

        # Convert to response format

        data = [

            RepoRateData(

                month=item['month'],

                month\_number=item['month\_number'],

                repo\_rate=item['repo\_rate'],

                reverse\_repo\_rate=item['reverse\_repo\_rate'],

                effective\_date=item['effective\_date']

            )

            for item in monthly\_data

        ]

        # Store in database

        conn = get\_connection()

        cursor = conn.cursor()

        for item in monthly\_data:

            cursor.execute('''

                INSERT INTO repo\_rates

                (year, month, month\_name, repo\_rate, reverse\_repo\_rate, effective\_date)

                VALUES (%s, %s, %s, %s, %s, %s)

                ON DUPLICATE KEY UPDATE

                repo\_rate = VALUES(repo\_rate),

                reverse\_repo\_rate = VALUES(reverse\_repo\_rate),

                created\_at = CURRENT\_TIMESTAMP

            ''', (

                year, item['month\_number'], item['month'],

                item['repo\_rate'], item['reverse\_repo\_rate'], item['effective\_date']

            ))

        # Log the update

        cursor.execute('''

            INSERT INTO repo\_rate\_update\_log (source\_url, records\_updated, status)

            VALUES (%s, %s, %s)

        ''', ("RBI Official Data", len(monthly\_data), "SUCCESS"))

        conn.commit()

        cursor.close()

        conn.close()

        # Count rate changes in the year

        rate\_changes = len(set(item.repo\_rate for item in data))

        return RepoRateResponse(

            year=year,

            data=data,

            last\_updated=datetime.now().isoformat(),

            source="Reserve Bank of India (RBI)",

            total\_changes=rate\_changes

        )

    except HTTPException:

        raise

    except Error as e:

        logger.error(f"Database error getting repo rate for year {year}: {str(e)}")

        raise HTTPException(status\_code=500, detail=f"Database error: {str(e)}")

    except Exception as e:

        logger.error(f"Error getting repo rate for year {year}: {str(e)}")

        raise HTTPException(status\_code=500, detail=f"Failed to fetch repo rate data for {year}: {str(e)}")

@router.post("/refresh")

async def refresh\_repo\_rate\_data(background\_tasks: BackgroundTasks):

    """Manually refresh repo rate data from RBI"""

    try:

        background\_tasks.add\_task(refresh\_all\_data)

        return {"message": "Data refresh initiated", "status": "in\_progress"}

    except Exception as e:

        logger.error(f"Error initiating data refresh: {str(e)}")

        raise HTTPException(status\_code=500, detail="Failed to initiate data refresh")

async def refresh\_all\_data():

    """Background task to refresh all repo rate data"""

    try:

        current\_rates = await rbi\_fetcher.fetch\_current\_rates()

        # Update data for all available years

        conn = get\_connection()

        cursor = conn.cursor()

        total\_records = 0

        for year in range(2020, 2026):

            monthly\_data = rbi\_fetcher.generate\_monthly\_data(year, current\_rates)

            for item in monthly\_data:

                cursor.execute('''

                    INSERT INTO repo\_rates

                    (year, month, month\_name, repo\_rate, reverse\_repo\_rate, effective\_date)

                    VALUES (%s, %s, %s, %s, %s, %s)

                    ON DUPLICATE KEY UPDATE

                    repo\_rate = VALUES(repo\_rate),

                    reverse\_repo\_rate = VALUES(reverse\_repo\_rate),

                    created\_at = CURRENT\_TIMESTAMP

                ''', (

                    year, item['month\_number'], item['month'],

                    item['repo\_rate'], item['reverse\_repo\_rate'], item['effective\_date']

                ))

                total\_records += 1

        # Log the background refresh

        cursor.execute('''

            INSERT INTO repo\_rate\_update\_log (source\_url, records\_updated, status)

            VALUES (%s, %s, %s)

        ''', ("Background Refresh - RBI Data", total\_records, "SUCCESS"))

        conn.commit()

        cursor.close()

        conn.close()

        logger.info(f"Successfully refreshed {total\_records} repo rate data records")

    except Error as e:

        logger.error(f"Database error in background refresh: {str(e)}")

        # Log the error

        try:

            conn = get\_connection()

            cursor = conn.cursor()

            cursor.execute('''

                INSERT INTO repo\_rate\_update\_log (source\_url, records\_updated, status, error\_message)

                VALUES (%s, %s, %s, %s)

            ''', ("Background Refresh - RBI Data", 0, "ERROR", str(e)))

            conn.commit()

            cursor.close()

            conn.close()

        except:

            pass

    except Exception as e:

        logger.error(f"Error in background refresh: {str(e)}")

# Health check endpoint

@router.get("/health")

async def health\_check():

    """Health check endpoint"""

    try:

        # Test database connection

        conn = get\_connection()

        cursor = conn.cursor()

        cursor.execute("SELECT 1")

        cursor.fetchone()

        cursor.close()

        conn.close()

        db\_status = "connected"

    except Exception as e:

        db\_status = f"error: {str(e)}"

    return {

        "status": "healthy",

        "service": "Repo Rate API",

        "timestamp": datetime.now().isoformat(),

        "database": db\_status

    }

**LOAN SIMULATOR**

from fastapi import APIRouter, Request

from fastapi.responses import JSONResponse

import json

import os

from math import pow

router = APIRouter()

INTEREST\_FILE = "loan\_interest.json"

# Load the admin-set interest rate

def load\_interest():

    if os.path.exists(INTEREST\_FILE):

        with open(INTEREST\_FILE, "r") as f:

            try:

                data = json.load(f)

                return float(data.get("interest\_rate", 8.0))

            except:

                return 8.0

    return 8.0

# Save updated interest rate

def save\_interest(rate: float):

    with open(INTEREST\_FILE, "w") as f:

        json.dump({"interest\_rate": rate}, f)

# API: Get current interest rate

@router.get("/interest")

async def get\_interest\_rate():

    return {"interest\_rate": load\_interest()}

# API: Admin updates interest rate

@router.post("/interest")

async def update\_interest\_rate(request: Request):

    try:

        body = await request.json()

        rate = float(body.get("interest\_rate"))

        if rate <= 0:

            return JSONResponse(content={"error": "Rate must be positive"}, status\_code=400)

        save\_interest(rate)

        return {"status": "success", "updated\_rate": rate}

    except:

        return JSONResponse(content={"error": "Invalid rate format"}, status\_code=400)

# API: Simulate EMI calculation based on amount, tenure, credit score

@router.post("/simulate")

async def simulate\_emi(request: Request):

    try:

        data = await request.json()

        amount = float(data.get("amount", 0))

        tenure = int(data.get("tenure", 0))

        credit\_score = int(data.get("credit\_score", 750))

        if amount <= 0 or tenure <= 0:

            return JSONResponse(content={"error": "Amount and tenure must be positive"}, status\_code=400)

        base\_rate = load\_interest()

        # Adjust rate based on credit score

        if credit\_score >= 800:

            rate = base\_rate - 0.5

        elif credit\_score >= 700:

            rate = base\_rate

        elif credit\_score >= 600:

            rate = base\_rate + 1.0

        else:

            rate = base\_rate + 2.0

        monthly\_rate = rate / 12 / 100

        emi = (amount \* monthly\_rate \* pow(1 + monthly\_rate, tenure)) / (pow(1 + monthly\_rate, tenure) - 1)

        total\_payment = emi \* tenure

        total\_interest = total\_payment - amount

        balance = amount

        schedule = []

        for month in range(1, tenure + 1):

            interest\_payment = balance \* monthly\_rate

            principal\_payment = emi - interest\_payment

            balance -= principal\_payment

            if balance < 0:

                balance = 0

            schedule.append({

                "month": month,

                "emi": round(emi, 2),

                "principal": round(principal\_payment, 2),

                "interest": round(interest\_payment, 2),

                "balance": round(balance, 2)

            })

        return {

            "rate\_used": rate,

            "emi": round(emi, 2),

            "total\_payment": round(total\_payment, 2),

            "total\_interest": round(total\_interest, 2),

            "schedule": schedule

        }

    except:

        return JSONResponse(content={"error": "Invalid request"}, status\_code=400)

loan\_router = router

**ANNOUNCEMENT BANNER**

import asyncio

import aiohttp

import json

import re

from datetime import datetime, timedelta

from typing import List, Dict, Optional

from fastapi import APIRouter, HTTPException, BackgroundTasks

from fastapi.responses import JSONResponse

from pydantic import BaseModel

from bs4 import BeautifulSoup

import redis

import logging

from urllib.parse import urljoin, urlparse

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

# Redis client for caching

try:

    redis\_client = redis.Redis(host='localhost', port=6379, db=0, decode\_responses=True)

except:

    redis\_client = None

    logger.warning("Redis not available, using in-memory cache")

# In-memory fallback cache

memory\_cache = {}

router = APIRouter(prefix="/api/announcements", tags=["announcements"])

class AnnouncementItem(BaseModel):

    title: str

    summary: str

    date: str

    source: str

    url: str

    category: str

    is\_new: bool = False

class AnnouncementScraper:

    def \_\_init\_\_(self):

        self.sources = {

            "RBI": {

                "url": "https://www.rbi.org.in/Scripts/BS\_PressReleaseDisplay.aspx",

                "notifications\_url": "https://www.rbi.org.in/Scripts/NotificationUser.aspx",

                "name": "Reserve Bank of India"

            },

            "EPFO": {

                "url": "https://www.epfindia.gov.in/site\_en/Whats\_New.php",

                "name": "Employees' Provident Fund Organisation"

            },

            "MOF": {

                "url": "https://www.finmin.nic.in/press\_room/2024",

                "name": "Ministry of Finance"

            },

            "PIB": {

                "url": "https://pib.gov.in/indexd.aspx",

                "name": "Press Information Bureau"

            }

        }

        self.keywords = [

            "interest rate", "repo rate", "reverse repo", "CRR", "SLR",

            "provident fund", "PF", "EPF", "pension", "gratuity",

            "loan", "credit", "lending rate", "deposit rate",

            "financial", "banking", "monetary policy",

            "deadline", "last date", "extension",

            "scheme", "subsidy", "relief"

        ]

    async def fetch\_page(self, session: aiohttp.ClientSession, url: str) -> Optional[str]:

        """Fetch webpage content"""

        try:

            headers = {

                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'

            }

            async with session.get(url, headers=headers, timeout=30) as response:

                if response.status == 200:

                    return await response.text()

                else:

                    logger.error(f"Failed to fetch {url}: Status {response.status}")

                    return None

        except Exception as e:

            logger.error(f"Error fetching {url}: {str(e)}")

            return None

    def is\_relevant(self, text: str) -> bool:

        """Check if content is relevant to financial/loan announcements"""

        text\_lower = text.lower()

        return any(keyword in text\_lower for keyword in self.keywords)

    def extract\_date(self, text: str) -> str:

        """Extract date from text"""

        try:

            # Common date patterns

            date\_patterns = [

                r'(\d{1,2}[/-]\d{1,2}[/-]\d{4})',

                r'(\d{1,2}\s+(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[a-z]\*\s+\d{4})',

                r'((?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[a-z]\*\s+\d{1,2},?\s+\d{4})',

            ]

            for pattern in date\_patterns:

                match = re.search(pattern, text, re.IGNORECASE)

                if match:

                    return match.group(1)

            return datetime.now().strftime('%d %b %Y')

        except:

            return datetime.now().strftime('%d %b %Y')

    async def scrape\_rbi(self, session: aiohttp.ClientSession) -> List[AnnouncementItem]:

        """Scrape RBI press releases and notifications"""

        announcements = []

        # Scrape press releases

        content = await self.fetch\_page(session, self.sources["RBI"]["url"])

        if content:

            soup = BeautifulSoup(content, 'html.parser')

            # Look for press release links

            for link in soup.find\_all('a', href=True):

                if link.get('href') and ('press' in link.get('href').lower() or 'notification' in link.get('href').lower()):

                    title = link.get\_text(strip=True)

                    if self.is\_relevant(title):

                        announcements.append(AnnouncementItem(

                            title=title[:100],

                            summary=title[:200],

                            date=self.extract\_date(title),

                            source="RBI",

                            url=urljoin(self.sources["RBI"]["url"], link['href']),

                            category="Monetary Policy"

                        ))

        # Scrape notifications

        content = await self.fetch\_page(session, self.sources["RBI"]["notifications\_url"])

        if content:

            soup = BeautifulSoup(content, 'html.parser')

            for row in soup.find\_all('tr'):

                cells = row.find\_all('td')

                if len(cells) >= 2:

                    title = cells[1].get\_text(strip=True) if len(cells) > 1 else ""

                    if self.is\_relevant(title):

                        link\_tag = cells[1].find('a') if len(cells) > 1 else None

                        url = urljoin(self.sources["RBI"]["notifications\_url"], link\_tag['href']) if link\_tag and link\_tag.get('href') else ""

                        announcements.append(AnnouncementItem(

                            title=title[:100],

                            summary=title[:200],

                            date=cells[0].get\_text(strip=True) if cells else self.extract\_date(title),

                            source="RBI",

                            url=url,

                            category="Banking Regulation"

                        ))

        return announcements[:10]  # Limit to 10 most recent

    async def scrape\_epfo(self, session: aiohttp.ClientSession) -> List[AnnouncementItem]:

        """Scrape EPFO updates"""

        announcements = []

        content = await self.fetch\_page(session, self.sources["EPFO"]["url"])

        if content:

            soup = BeautifulSoup(content, 'html.parser')

            # Look for news/updates sections

            for item in soup.find\_all(['div', 'li', 'tr'], class\_=re.compile(r'news|update|announcement', re.I)):

                text = item.get\_text(strip=True)

                if self.is\_relevant(text):

                    link = item.find('a')

                    url = urljoin(self.sources["EPFO"]["url"], link['href']) if link and link.get('href') else ""

                    announcements.append(AnnouncementItem(

                        title=text[:100],

                        summary=text[:200],

                        date=self.extract\_date(text),

                        source="EPFO",

                        url=url,

                        category="Provident Fund"

                    ))

        return announcements[:5]

    async def scrape\_mof(self, session: aiohttp.ClientSession) -> List[AnnouncementItem]:

        """Scrape Ministry of Finance press releases"""

        announcements = []

        content = await self.fetch\_page(session, self.sources["MOF"]["url"])

        if content:

            soup = BeautifulSoup(content, 'html.parser')

            for link in soup.find\_all('a', href=True):

                title = link.get\_text(strip=True)

                if self.is\_relevant(title):

                    announcements.append(AnnouncementItem(

                        title=title[:100],

                        summary=title[:200],

                        date=self.extract\_date(title),

                        source="Ministry of Finance",

                        url=urljoin(self.sources["MOF"]["url"], link['href']),

                        category="Government Policy"

                    ))

        return announcements[:5]

    async def scrape\_all(self) -> List[AnnouncementItem]:

        """Scrape all sources"""

        all\_announcements = []

        async with aiohttp.ClientSession() as session:

            tasks = [

                self.scrape\_rbi(session),

                self.scrape\_epfo(session),

                self.scrape\_mof(session)

            ]

            results = await asyncio.gather(\*tasks, return\_exceptions=True)

            for result in results:

                if isinstance(result, list):

                    all\_announcements.extend(result)

                else:

                    logger.error(f"Scraping error: {result}")

        # Sort by date (newest first) and mark new items

        now = datetime.now()

        for announcement in all\_announcements:

            try:

                # Simple check for "new" items (last 24 hours)

                announcement.is\_new = True  # Simplified for demo

            except:

                pass

        # Remove duplicates based on title similarity

        unique\_announcements = []

        seen\_titles = set()

        for announcement in all\_announcements:

            title\_key = re.sub(r'[^\w\s]', '', announcement.title.lower())[:50]

            if title\_key not in seen\_titles:

                seen\_titles.add(title\_key)

                unique\_announcements.append(announcement)

        return sorted(unique\_announcements, key=lambda x: x.date, reverse=True)[:20]

# Global scraper instance

scraper = AnnouncementScraper()

def cache\_set(key: str, value: str, expire: int = 3600):

    """Set cache value"""

    if redis\_client:

        redis\_client.setex(key, expire, value)

    else:

        memory\_cache[key] = {"value": value, "expire": datetime.now() + timedelta(seconds=expire)}

def cache\_get(key: str) -> Optional[str]:

    """Get cache value"""

    if redis\_client:

        return redis\_client.get(key)

    else:

        if key in memory\_cache:

            if datetime.now() < memory\_cache[key]["expire"]:

                return memory\_cache[key]["value"]

            else:

                del memory\_cache[key]

        return None

async def update\_announcements():

    """Background task to update announcements"""

    try:

        logger.info("Starting announcement update...")

        announcements = await scraper.scrape\_all()

        # Cache the results for 15 minutes

        cache\_data = json.dumps([announcement.dict() for announcement in announcements])

        cache\_set("announcements", cache\_data, 900)  # 15 minutes

        logger.info(f"Updated {len(announcements)} announcements")

        return announcements

    except Exception as e:

        logger.error(f"Error updating announcements: {str(e)}")

        return []

@router.get("/", response\_model=List[AnnouncementItem])

async def get\_announcements(background\_tasks: BackgroundTasks):

    """Get latest announcements"""

    try:

        # Try to get from cache first

        cached\_data = cache\_get("announcements")

        if cached\_data:

            try:

                announcements\_data = json.loads(cached\_data)

                return [AnnouncementItem(\*\*item) for item in announcements\_data]

            except:

                pass

        # If no cache, trigger background update and return empty or sample data

        background\_tasks.add\_task(update\_announcements)

        # Return sample data for immediate response

        return [

            AnnouncementItem(

                title="Loading latest government announcements...",

                summary="Fetching real-time data from official sources",

                date=datetime.now().strftime('%d %b %Y'),

                source="System",

                url="#",

                category="System",

                is\_new=True

            )

        ]

    except Exception as e:

        logger.error(f"Error in get\_announcements: {str(e)}")

        raise HTTPException(status\_code=500, detail="Failed to fetch announcements")

@router.post("/refresh")

async def refresh\_announcements():

    """Manually refresh announcements"""

    try:

        announcements = await update\_announcements()

        return JSONResponse(content={

            "status": "success",

            "message": f"Refreshed {len(announcements)} announcements",

            "count": len(announcements)

        })

    except Exception as e:

        logger.error(f"Error refreshing announcements: {str(e)}")

        raise HTTPException(status\_code=500, detail="Failed to refresh announcements")

@router.get("/stats")

async def get\_stats():

    """Get announcement statistics"""

    try:

        cached\_data = cache\_get("announcements")

        if cached\_data:

            announcements\_data = json.loads(cached\_data)

            # Count by source

            source\_counts = {}

            category\_counts = {}

            new\_count = 0

            for item in announcements\_data:

                source = item.get("source", "Unknown")

                category = item.get("category", "General")

                is\_new = item.get("is\_new", False)

                source\_counts[source] = source\_counts.get(source, 0) + 1

                category\_counts[category] = category\_counts.get(category, 0) + 1

                if is\_new:

                    new\_count += 1

            return {

                "total": len(announcements\_data),

                "new": new\_count,

                "sources": source\_counts,

                "categories": category\_counts,

                "last\_updated": datetime.now().isoformat()

            }

        else:

            return {

                "total": 0,

                "new": 0,

                "sources": {},

                "categories": {},

                "last\_updated": None

            }

    except Exception as e:

        logger.error(f"Error getting stats: {str(e)}")

        raise HTTPException(status\_code=500, detail="Failed to get statistics")

# Startup event to load initial data

@router.on\_event("startup")

async def startup\_event():

    """Load initial announcements on startup"""

    asyncio.create\_task(update\_announcements())

**EXPENCE TRACKER**

from fastapi import APIRouter, Query, HTTPException

from fastapi.responses import StreamingResponse

from pydantic import BaseModel, Field, field\_validator

from typing import List, Optional, Dict

import datetime

import io

from reportlab.lib.pagesizes import letter

from reportlab.platypus import SimpleDocTemplate, Paragraph, Table, TableStyle, Spacer

from reportlab.lib import colors

from reportlab.lib.styles import getSampleStyleSheet

from db import get\_connection

router = APIRouter(prefix="/api/expenses", tags=["Expenses"])

# Expense model for API output

class Expense(BaseModel):

    id: int

    date: datetime.date

    amount: float

    category: str

    merchant: Optional[str] = None

    notes: Optional[str] = None

# Model for creating new expense

class ExpenseCreate(BaseModel):

    date: str

    amount: float = Field(..., gt=0)

    category: str = Field(..., min\_length=1)

    merchant: Optional[str] = None

    notes: Optional[str] = None

    @field\_validator("date")

    def parse\_date(cls, v):

        try:

            return datetime.datetime.strptime(v, "%d-%m-%Y").date()

        except ValueError:

            raise ValueError("date must be in DD-MM-YYYY format")

@router.post("/", response\_model=Expense)

def add\_expense(payload: ExpenseCreate):

    date\_obj = payload.date

    conn = get\_connection()

    try:

        with conn.cursor() as cursor:

            sql = """

                INSERT INTO expenses (date, amount, category, merchant, notes)

                VALUES (%s, %s, %s, %s, %s)

            """

            cursor.execute(sql, (

                date\_obj.strftime("%Y-%m-%d"),

                payload.amount,

                payload.category,

                payload.merchant,

                payload.notes

            ))

            conn.commit()

            expense\_id = cursor.lastrowid

    finally:

        conn.close()

    return Expense(

        id=expense\_id,

        date=date\_obj,

        amount=payload.amount,

        category=payload.category,

        merchant=payload.merchant,

        notes=payload.notes

    )

@router.get("/", response\_model=List[Expense])

def list\_expenses(

    start\_date: Optional[datetime.date] = Query(None),

    end\_date: Optional[datetime.date] = Query(None),

    category: Optional[str] = Query(None)

):

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            sql = "SELECT \* FROM expenses"

            params = []

            if start\_date:

                sql += " AND date >= %s"

                params.append(start\_date.strftime("%Y-%m-%d"))

            if end\_date:

                sql += " AND date <= %s"

                params.append(end\_date.strftime("%Y-%m-%d"))

            if category:

                sql += " AND category = %s"

                params.append(category)

            sql += " ORDER BY date DESC"

            cursor.execute(sql, tuple(params))

            rows = cursor.fetchall()

            for row in rows:

                row["date"] = row["date"] if isinstance(row["date"], datetime.date) else row["date"].date()

            return [Expense(\*\*row) for row in rows]

    finally:

        conn.close()

@router.get("/summary", response\_model=Dict[str, object])

def monthly\_summary(

    month: int = Query(datetime.date.today().month),

    year: int = Query(datetime.date.today().year)

):

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            sql = """

                SELECT category, SUM(amount) AS total

                FROM expenses

                WHERE MONTH(date) = %s AND YEAR(date) = %s

                GROUP BY category

            """

            cursor.execute(sql, (month, year))

            breakdown\_list = cursor.fetchall()

            breakdown = {row["category"]: float(row["total"] or 0) for row in breakdown\_list}

            cursor.execute(

                "SELECT COUNT(\*) as count, SUM(amount) as total FROM expenses WHERE MONTH(date)=%s AND YEAR(date)=%s",

                (month, year)

            )

            stats = cursor.fetchone()

            transactions = stats["count"] or 0

            total = float(stats["total"] or 0.0)

            categories\_count = len(breakdown)

            top = max(breakdown.items(), key=lambda x: x[1], default=("", 0.0))

    finally:

        conn.close()

    return {

        "month": f"{year}-{month:02d}",

        "total": total,

        "breakdown": breakdown,

        "transactions": transactions,

        "categories\_count": categories\_count,

        "top\_category": {"name": top[0], "amount": top[1]}

    }

# Export monthly summary as CSV

@router.get("/summary/export")

def export\_summary(

    year: int = Query(datetime.date.today().year),

    month: int = Query(datetime.date.today().month)

):

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("""

                SELECT category, SUM(amount) AS total

                FROM expenses

                WHERE MONTH(date) = %s AND YEAR(date) = %s

                GROUP BY category

            """, (month, year))

            breakdown\_list = cursor.fetchall()

    finally:

        conn.close()

    output = io.StringIO()

    output.write("Category,Amount\n")

    for row in breakdown\_list:

        output.write(f"{row['category']},₹{row['total']:.2f}\n")

    output.seek(0)

    filename = f"expense\_summary\_{year}\_{month:02d}.csv"

    return StreamingResponse(

        output,

        media\_type="text/csv",

        headers={"Content-Disposition": f"attachment; filename=\"{filename}\""}

    )

# Export all expenses as CSV

@router.get("/export")

def export\_all\_expenses():

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute("SELECT \* FROM expenses ORDER BY date DESC")

            rows = cursor.fetchall()

    finally:

        conn.close()

    output = io.StringIO()

    output.write("ID,Date,Category,Amount,Merchant,Notes\n")

    for row in rows:

        date\_str = row["date"].strftime("%Y-%m-%d") if isinstance(row["date"], datetime.date) else str(row["date"])

        amt = f"₹{row['amount']:.2f}"

        merchant = row["merchant"] or ""

        notes = row["notes"] or ""

        output.write(f"{row['id']},{date\_str},{row['category']},{amt},{merchant},{notes}\n")

    output.seek(0)

    filename = "all\_expenses.csv"

    return StreamingResponse(

        output,

        media\_type="text/csv",

        headers={"Content-Disposition": f"attachment; filename=\"{filename}\""}

    )

# Generate PDF report for a month

@router.get("/report/pdf")

def report\_pdf(

    month: int = Query(datetime.date.today().month),

    year: int = Query(datetime.date.today().year)

):

    conn = get\_connection()

    try:

        with conn.cursor(dictionary=True) as cursor:

            cursor.execute(

                "SELECT \* FROM expenses WHERE MONTH(date)=%s AND YEAR(date)=%s ORDER BY date",

                (month, year)

            )

            items = cursor.fetchall()

            cursor.execute(

                "SELECT category, SUM(amount) as total FROM expenses WHERE MONTH(date)=%s AND YEAR(date)=%s GROUP BY category",

                (month, year)

            )

            breakdown\_list = cursor.fetchall()

    finally:

        conn.close()

    total = sum(e["amount"] for e in items)

    count = len(items)

    avg = total / count if count else 0

    breakdown = {row["category"]: row["total"] for row in breakdown\_list}

    top = max(breakdown.items(), key=lambda x: x[1], default=("", 0))

    buffer = io.BytesIO()

    doc = SimpleDocTemplate(

        buffer,

        pagesize=letter,

        leftMargin=60, rightMargin=60,

        topMargin=100, bottomMargin=60

    )

    styles = getSampleStyleSheet()

    elems = []

    elems.append(Paragraph("<para alignment='center'><b>Expense Tracker Report</b></para>", styles['Title']))

    elems.append(Spacer(1, 24))

    summary\_data = [

        ["Metric",       "Value"],

        ["Total Spent",   f"Rs. {total:.2f}"],

        ["Transactions",  str(count)],

        ["Average Spend", f"Rs. {avg:.2f}"],

        ["Top Category",  f"{top[0]} (Rs. {top[1]:.2f})"]

    ]

    summary\_tbl = Table(summary\_data, colWidths=[200, 200], hAlign='CENTER')

    summary\_tbl.setStyle(TableStyle([

        ('BOX',       (0,0), (-1,-1), 1.5, colors.black),

        ('GRID',      (0,0), (-1,-1), 0.5, colors.grey),

        ('BACKGROUND',(0,0), (-1,0), colors.lightgrey),

        ('ALIGN',     (0,0), (-1,-1), 'CENTER'),

        ('FONTNAME',  (0,0), (-1,0), 'Helvetica-Bold'),

    ]))

    elems.append(summary\_tbl)

    elems.append(Spacer(1, 20))

    advice\_text = f"Consider setting a monthly budget for {top[0]} to manage costs effectively."

    elems.append(Paragraph(f"<para alignment='center'><i>{advice\_text}</i></para>", styles['Normal']))

    elems.append(Spacer(1, 20))

    elems.append(Paragraph("<b>Category Breakdown</b>", styles['Heading3']))

    brk\_data = [['Category','Amount']] + [[c, f"Rs. {amt:.2f}"] for c, amt in breakdown.items()]

    brk\_tbl = Table(brk\_data, colWidths=[200,200], hAlign='CENTER')

    brk\_tbl.setStyle(TableStyle([

        ('BOX',       (0,0), (-1,-1), 1.5, colors.black),

        ('GRID',      (0,0), (-1,-1), 0.5, colors.grey),

        ('BACKGROUND',(0,0), (-1,0), colors.lightgrey),

        ('ALIGN',     (0,0), (-1,-1), 'CENTER'),

        ('FONTNAME',  (0,0), (-1,0), 'Helvetica-Bold'),

    ]))

    elems.append(brk\_tbl)

    elems.append(Spacer(1, 20))

    elems.append(Paragraph("<b>Detailed Expenses</b>", styles['Heading3']))

    det\_data = [['Date','Category','Amount']] + [

        [

            (e["date"].strftime("%Y-%m-%d") if isinstance(e["date"], datetime.date) else str(e["date"])),

            e["category"],

            f"Rs. {e['amount']:.2f}"

        ] for e in items

    ]

    det\_tbl = Table(det\_data, colWidths=[120,180,100], hAlign='CENTER')

    det\_tbl.setStyle(TableStyle([

        ('BOX',       (0,0), (-1,-1), 1.5, colors.black),

        ('GRID',      (0,0), (-1,-1), 0.5, colors.grey),

        ('BACKGROUND',(0,0), (-1,0), colors.lightgrey),

        ('ALIGN',     (0,0), (-1,-1), 'CENTER'),

        ('FONTNAME',  (0,0), (-1,0), 'Helvetica-Bold'),

    ]))

    elems.append(det\_tbl)

    def draw\_border\_header\_footer(cnv, doc):

        width, height = letter

        cnv.saveState()

        cnv.setLineWidth(2)

        cnv.rect(doc.leftMargin-10, doc.bottomMargin-10,

                 width - (doc.leftMargin-10) - (doc.rightMargin-10),

                 height - (doc.topMargin-10) - (doc.bottomMargin-10))

        footer\_text = f"Generated: {datetime.date.today().isoformat()} | Report"

        cnv.setFont('Helvetica-Oblique', 8)

        cnv.drawCentredString(width/2, doc.bottomMargin - 20, footer\_text)

        cnv.restoreState()

    doc.build(

        elems,

        onFirstPage=draw\_border\_header\_footer,

        onLaterPages=draw\_border\_header\_footer

    )

    buffer.seek(0)

    return StreamingResponse(

        buffer,

        media\_type='application/pdf',

        headers={'Content-Disposition': f'attachment; filename="expense\_report\_{year}\_{month:02d}.pdf"'}

    )